



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Ph.D. DISSERTATION

Dynamic Behavior Specification and Design Space Exploration for Real-time Embedded Systems

실시간 임베디드 시스템을 위한 동적 행위 명세 및
설계 공간 탐색 기법

August 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hanwoong Jung

Abstract

As the number of processors in a chip increases, and more functions are integrated, the system status will change dynamically due to various factors such as the workload variation, QoS requirement, and unexpected component failure. On the other hand, computation-complexity of user applications is also steadily increasing; video and graphics applications are two major driving forces in smart mobile devices, which define the main application domain of interest in this dissertation. So, a systematic design methodology is highly required to implement such complex systems which contain dynamically changed behavior as well as computation-intensive workload that can be parallelized.

A model-based approach is one of representative approaches for parallel embedded software development. Especially, HOPES framework is proposed which is a design environment for parallel embedded software supporting the overall design steps: system specification, performance estimation, design space exploration, and automatic code generation. Distinguished from other design environments, it introduces a novel concept of “programming platform”, called CIC (Common Intermediate Code) that can be understood as a generic execution model of heterogeneous multiprocessor architecture. The CIC task model is based on a process network model, but it can be refined to the SDF (Synchronous Data Flow) model, since it has a very desirable features for static

analyzability as well as parallel processing. However, the SDF model has a typical weakness of expression capability, especially for the system-level specification and dynamically changed behavior of an application.

To overcome this weakness, in this dissertation, we propose an extended CIC task model based on dataflow and FSM models to specify the dynamic behavior of the system distinguishing inter- and intra-application dynamism. At the top-level, each application is specified by a dataflow task and the dynamic behavior is modeled as a control task that supervises the execution of applications. Inside a dataflow task, it specifies the dynamic behavior using a similar way as FSM-based SADF; an SDF task may have multiple behaviors and a tabular specification of an FSM, called MTM (Mode Transition Machine), describes the mode transition rules for the SDF graph. We call it to MTM-SDF model which is classified as multi-mode dataflow models in the dissertation. It assumes that an application has a finite number of behaviors (or modes) and each behavior (mode) is represented by an SDF graph. It enables us to perform compile-time scheduling of each graph to maximize the throughput varying the number of allocated processors, and store the scheduling information.

Also, a multiprocessor scheduling technique is proposed for a multi-mode dataflow graph. While there exist several scheduling techniques for multi-mode dataflow models, no one allows task migration between modes. By observing that the resource requirement can be

additionally reduced if task migration is allowed, we propose a multiprocessor scheduling technique of a multi-mode dataflow graph considering task migration between modes. Based on a genetic algorithm, the proposed technique schedules all SDF graphs in all modes simultaneously to minimize the resource requirement. To satisfy the throughput constraint, the proposed technique calculates the actual throughput requirement of each mode and the output buffer size for tolerating throughput jitter.

For the specified task graph and scheduling results, the CIC translator generates parallelized code for the target architecture. Therefore the CIC translator is extended to support extended features of the CIC task model. In application-level, it is extended to support multiprocessor code generation for an MTM-SDF graph considering the given static scheduling results. Also, multiprocessor code generation of four different scheduling policies are supported for an MTM-SDF graph: fully-static, self-timed, static-assignment, and fully-dynamic. In system-level, the CIC translator is extended to support code generation for implementation of system request APIs and data structures for the static scheduling results and configurable task parameters.

Through preliminary experiments with a multi-mode multimedia terminal example, the viability of the proposed methodology is verified.

Keywords: model-based design methodology, synchronous dataflow, multi-mode dataflow, multiprocessor scheduling, mode transition delay, automatic code synthesis, parallel code generation

Student number: 2012-30230

Contents

Contents v

List of Figures ix

List of Tables xiii

Chapter 1	Introduction	1
1.1	Motivation	1
1.2	Contribution.....	7
1.3	Dissertation organization.....	9
Chapter 2	Background.....	10
2.1	Related work.....	10
2.1.1	Compiler-based approach.....	10
2.1.2	Language-based approach.....	11

2.1.3	Model-based approach	15
2.2	HOPES framework.....	19
2.3	Common Intermediate Code (CIC) Model.....	21
Chapter 3	Dynamic Behavior Specification.....	26
3.1	Problem definition.....	26
3.1.1	System-level dynamic behavior	26
3.1.2	Application-level dynamic behavior	27
3.2	Related work.....	28
3.3	Motivational example.....	31
3.4	Control task specification for system-level dynamism	33
3.4.1	Internal specification	33
3.4.2	Action scripts.....	38
3.5	MTM-SDF specification for application-level dynamism	44
3.5.1	MTM specification.....	44
3.5.2	Task graph specification.....	45
3.5.3	Execution semantic of an MTM-SDF graph.....	46
Chapter 4	Multiprocessor Scheduling of an Multi-mode Dataflow	
Graph	50	

4.1	Related work.....	51
4.2	Motivational example.....	56
4.2.1	Throughput requirement calculation considering mode transition delay	56
4.2.2	Task migration between mode transition	58
4.3	Problem definition.....	61
4.4	Throughput requirement analysis.....	65
4.4.1	Mode transition delay.....	66
4.4.2	Arrival curves of the output buffer.....	70
4.4.3	Buffer size determination	71
4.4.4	Throughput requirement analysis.....	73
4.5	Proposed MMDF scheduling framework.....	75
4.5.1	Optimization problem	75
4.5.2	GA configuration.....	76
4.5.3	Fitness function	78
4.5.4	Local optimization technique	79
4.6	Experimental results	81
4.6.1	MMDF scheduling technique.....	83
4.6.2	Scalability of the Proposed Framework	88

Chapter 5	Multiprocessor Code Generation for the Extended CIC	
Model		89
5.1	CIC translator	89
5.2	Code generation for application-level dynamism	91
5.2.1	Function call-style code generation (fully-static, self-timed)	94
5.2.2	Thread-style code generation (static-assignment, fully-dynamic)	98
5.3	Code generation for system-level dynamism	101
5.4	Experimental results	105
Chapter 6	Conclusion and Future Work	107
Bibliography		109

List of Figures

Figure 2-1. Usage of OpenMP directives.....	12
Figure 2-2. Usage of MPI APIs.....	13
Figure 2-3. Usage of OpenCL APIs	14
Figure 2-4. An SDF graph and schedule example.....	16
Figure 2-5. HOPES design flow	19
Figure 2-6. CIC task graph.....	21
Figure 2-7. CIC task code template	23
Figure 3-1. System-level dynamic behavior	26
Figure 3-2. Application-level dynamic behavior	27
Figure 3-3. A multi-mode multimedia terminal example.....	31
Figure 3-4. FSM in the control task in Figure 3-3	37
Figure 3-5. Example code for control task triggering by system external events....	39
Figure 3-6. Example code for control task triggering by system internal events.....	40

Figure 3-7. Example code for control task triggering by timeout events	40
Figure 3-8. Example code for execution status control.....	41
Figure 3-9. Example code for task parameter control.....	41
Figure 3-10. Example code for task mode control.....	42
Figure 3-11. Example code for timing requirement control.....	43
Figure 3-12. Mode transition machine specification.....	44
Figure 3-13. CIC task code skeleton in an MTM-SDF graph.....	46
Figure 3-14. Timeline of the MTM-SDF graph execution.....	47
Figure 3-15. An MTM-SDF graph specification of H.264 decoder application	48
Figure 4-1. Extracted behavior of the entire system from the specification	50
Figure 4-2. Motivational example of throughput requirement calculation considering the mode transition delay	56
Figure 4-3. Motivational example of task migration	58
Figure 4-4. An MMDF graph example	61
Figure 4-5. An MMDF graph example and static scheduling result.....	66
Figure 4-6. Mode transition delay between static schedules of mode m1 and m2 in Figure 4-5 (b)	67
Figure 4-7. Arrival curves for input and output streams in the output buffer	70
Figure 4-8. Overlapped schedule sequences among modes (ma \rightarrow mb \rightarrow mc)	75

Figure 4-9. The overall GA framework.....	77
Figure 4-10. Chromosome structure	77
Figure 4-11. Without processor renaming, every task should be migrated when mode transition occurs. If PE0 in mode 0 is renamed to PE2 in mode 1, PE1 to PE0, and PE2 to PE1, no task migration is required.....	79
Figure 4-12. Pseudo code of the processor renaming heuristic	80
Figure 4-13. MMDF graph examples used in experiments.....	82
Figure 4-14. Comparison results in terms of the number of processors: <i>Base</i> , <i>Fixed</i> , <i>Blocked</i> , and <i>Proposed</i>	86
Figure 4-15. Experimental result for the scalability property	88
Figure 5-1. Code structure of the automatically generated code.....	89
Figure 5-2. Generated code for the specified MTM (<i>task_name.mtm</i>).....	91
Figure 5-3. Run-time execution scenario for each scheduling policy.....	93
Figure 5-4. Data structure for MTM-SDF graphs and task execution routine considering the static scheduling result in the self-timed policy	95
Figure 5-5. Task execution routine considering the static scheduling result in the fully-static policy	96
Figure 5-6. Task execution routine in the function call style	97
Figure 5-7. Data structure for MTM-SDF graphs in the thread style	98
Figure 5-8. Task execution routine of an MTM-SDF graph in thread style.....	100

Figure 5-9. Implementation of system request APIs for task execution control	102
Figure 5-10. Implementation of system request APIs for task constraint control..	103
Figure 5-11. Code structure of the automatically generated code.....	104
Figure 5-12. Comparison results between DSE results and four code generation policies for a multi-mode multimedia terminal example	105

List of Tables

Table 2-1. Category of widely-used models	15
Table 3-1. Use cases of the multi-mode multimedia terminal system in Figure 3-3	32
Table 3-2. Control APIs in the proposed model	38
Table 3-3. Control APIs for an MTM-SDF graph.....	46
Table 4-1. Comparison of various MoCs supporting dynamic behavior specification	51
Table 4-2. Configuration of the GA framework.....	81
Table 4-3. Four approaches used in experiments	84
Table 4-4. Configurations for experiments	85
Table 4-5. Experimental results of <i>Proposed</i> approach	87
Table 5-1. Scheduling policies for a dataflow graph.....	92

Chapter 1 Introduction

1.1 Motivation

Incessant semiconductor technology improvement allows us to integrate tens or hundreds of processors in a single chip and make a multiprocessor systems-on-chip (MPSoC) [1], or a multi-/many-core platform. As the power consumption becomes the limiting factor of the computing power and an application-specific processing element provides higher performance per power consumption than a general purpose processor core, a future multi-core processor will likely consist of heterogeneous processing elements that are tailored for applications. On the other hand, computation-complexity of user applications is also steadily increasing; video and graphics applications are two major driving forces in smart mobile devices, which define the main application domain of interest in this dissertation. To cope with the increasing computation workload, it becomes popular to equip a many-core accelerator to utilize the parallelism of an application maximally.

Also, in such a system, the system status may change dynamically due to various factors. At the system level, the set of applications running concurrently may change according to user requests. At the application level, the workload of a task may vary depending on the dynamically varying operation mode. At the operating system level, we may want to change the resource assignment to the applications based on various factors such as QoS (quality-of-service) requirement, power budget, and heat dissipation. At the hardware level, hardware resource availability might vary since hardware components may experience intermittent or permanent

failure as the technology scaling continues [2]. How to support such dynamism in the heterogeneous multiprocessor platform is the main theme of this dissertation.

Model-based design methodology [3] is widely accepted for embedded system design since it enables us to cope with ever increasing system complexity by maximizing the benefit of abstraction. As an algorithm specification model, this dissertation adopts a coarse-grain dataflow model which is suitable for specifying signal processing or streaming applications. In a dataflow graph, a node presents a function module and an arc represents the flow of data samples (or tokens) through the FIFO channel between two end nodes. When a node is invoked, it consumes a specified number of data samples (called *sample rate*) from each input arc and produces a specified number of samples to each output arc. A node becomes executable when all input arcs have as many data samples as the specified sample rates. If the sample rate is a fixed integer number which does not change at run-time, the dataflow graph is called a synchronous dataflow (SDF) graph [4]. An SDF graph is called consistent when there exists a schedule of node executions that does not change the number of samples on all FIFO channels. Such a schedule that consists of the minimum number of node executions is called an *iteration* of the SDF graph.

The SDF model expresses explicitly various kinds of potential parallelism of an application, which is desirable for multicore programming. If there is no data dependency between nodes, they can be performed concurrently, which corresponds to task-level parallelism. When an application processes a stream of data samples, each task also processes a stream of input data samples. In case a task has no internal state, multiple copies of the task can be instantiated to process multiple data samples in parallel to exploit data parallelism of the application [5]. In addition, the task graph itself can be invoked multiple times in a pipelined fashion to process the

stream of input data samples to maximize the throughput performance, which corresponds to temporal parallelism [6].

But the SDF model has a severe restriction to be used for modern embedded applications. It cannot express the dynamic behavior of an application, while modern embedded applications become more complex with dynamic behavior changes at runtime.

The first problem to solve is how to specify the system behavior that may change dynamically. We distinguish two kinds of dynamism for behavior specification. The first one corresponds to the case where an application has a set of multiple operation modes. For example, a video decoder behaves differently depending on the type of encoded input frame: I, P, or B type. Or an application may use different algorithms depending on the given resource and the timing budget. To express this kind of application-level dynamism, several extensions have been proposed to the SDF model, including scenario-aware dataflow (SADF) [7], parameterized SDF (PSDF) [8], and so on. They assume that an application has a finite number of behaviors (or modes) and each behavior (mode) is represented by an SDF graph. Similarly, we use a hybrid model of a dataflow model and an FSM, called MTM (Mode Transition Machine)-SDF model, almost identical to FSM-SADF (FSM-based Scenario-Aware Data Flow) [10]. In those extended models, an application consists of a finite number of operation modes and each mode is specified by an SDF graph. The mode transition is specified by a mode transition machine which is a tabular specification of an FSM that describes the mode transition rules for the task graph. Since the number of samples produced or consumed at each port per node execution is fixed in the SDF model, we can perform static mapping and/or scheduling of an SDF graph for each mode of operation. Through the static scheduling, we can estimate

the performance and the resource requirement of the application. We categorize such SDF-based extended models as a multi-mode dataflow (MMDF) model and define the MMDF model that can be implemented by any specific extension.

The second kind of dynamic behavior occurs when a set of applications running concurrently varies; a system has multiple use cases. A use-case of a system is defined by a set of applications running concurrently. The change of use cases is specified by an FSM in the proposed technique. To this end, we distinguish a control task from a normal computation task as PeaCE specification [11][12] and a control task defines its behavior with an FSM that controls the transition between use cases. Thus, a control task plays the role of a supervisory task for a set of applications that it controls. We also specify the timing requirement of each application explicitly inside the control task as part of the initial specification with the aim to make the initial specification independent of the hardware platform in functionality and timing [13][14].

In summary, the proposed specification method is hierarchical. At the top-level, there are two types of tasks, control and computation tasks, in the task graph. A computation task at the top-level may contain a subgraph inside that is based on the MTM-SDF model.

The second problem to solve is how to schedule a set of applications in each use case. In an MMDF graph, we assume that the mode transition is made at the iteration boundary while the mode switching decision can be made any time during the execution. In each mode, the associated SDF graph is scheduled statically to satisfy the throughput constraint. If the static schedule is followed at run-time, output samples will be produced periodically. When mode switching occurs, however, we may have to pay extra overhead of mode transition, called *mode*

transition delay, which affects the throughput performance of the application. Thus we propose an analytical formula to compute the mode transition delay and investigate how the mode transition delay affects the throughput performance when the schedule of each mode is given.

While there exist several techniques [71][74] to schedule an MMDF graph with considering the mode transition delay, they assume that each task is mapped to the same processor throughout all modes. On the other hand, we propose a scheduling technique that does not have such an assumption, observing that the number of required processors can be reduced if a task can be mapped onto different processors among modes. Suppose that a task is mapped onto different processors between two modes. Then, we have to pay extra overhead to migrate the tasks, called *task migration delay*, which should be included in the computation of mode transition delay.

Therefore, in this dissertation, we propose a multiprocessor scheduling technique of a multi-mode dataflow graph considering the mode transition delay conservatively. Our scheduling objective is to minimize the number of processors while satisfying the overall throughput constraint with considering the mode transition delay. Also, we propose a formulation to compute the output buffer size to mitigate the time fluctuation of output results.

Lastly, multiprocessor code generation technique is proposed for the extended task model. In application-level, the automatic code generator synthesizes multiprocessor code for an MTM-SDF graph considering the static scheduling results. Especially, it supports multiprocessor code generation for four different scheduling policies: fully-static, self-timed, static-assignment, and fully-dynamic. In system-level, it generates target dependent code such as thread management and

channel management as well as control functions and data structures for static scheduling results and configurable properties of computation tasks.

In summary, this dissertation covers the overall design steps of the model-based design approach. It extends the HOPES framework [15] to support dynamic behavior of the system in each design step: specification, analysis and code generation.

1.2 Contribution

The contributions of this dissertation can be summarized as follows.

- 1) We propose the extended task model to specify dynamic behavior of the system based on formal models.
 - A. For system-level dynamic behavior specification, a control task is proposed which plays the role of a supervisory task for a set of applications that it controls.
 - B. For application-level dynamic behavior specification, we propose an MTM-SDF model in which each mode of operation is specified by an SDF graph and mode transition is expressed by an FSM model.
 - C. To keep static analyzability, the proposed task model is extended based on formal models such as dataflow and finite state machine models for both system-level and application-level dynamic behavior specification.
- 2) We propose a multiprocessor scheduling technique for a multi-mode dataflow graph considering mode transition delay.
 - A. To calculate the actual throughput of a multi-mode dataflow graph, we formulate the mode transition delay among modes.
 - B. To minimize the resource requirement, we allow task migration among modes, and the task migration delay is taken into account for the mode transition delay.
 - C. For the given throughput constraint, we calculate actual throughput

requirement of each mode and the required output buffer size based on arrival curves.

- D. We find a static schedule of an MMDF graph which minimizes the resource requirement.
- 3) We support multiprocessor code generation for the extended task graph considering the static scheduling results.
- A. In system-level, the code synthesizer generates target dependent code such as thread management and channel management as well as control functions and data structures for static scheduling results and configurable task parameters.
 - B. In application-level, the code synthesizer generates multiprocessor code for four different scheduling policies: fully-static, self-timed, static-assignment, and fully- dynamic policies.
- 4) The overall design flow from specification to automatic code generation is implemented and integrated to HOPES framework

1.3 Dissertation organization

The rest of the dissertation is organized as follows: The base task model is introduced in Chapter 2, and details of the extensions for dynamic behavior specification are introduced in Chapter 3. Multiprocessor scheduling technique of a multi-mode dataflow graph is proposed in Chapter 4, and multiprocessor code generation technique for the extended task model is introduced in Chapter 5. Lastly, a conclusion will be presented.

Chapter 2 Background

2.1 Related work

As the more cores and processors are integrated into a single-chip and the complexity of applications increases, extensive efforts are being made to develop parallel embedded software for multi-core (or processor) systems. There exist various parallel programming models and they can be classified to several categories depending on their approaches [16].

2.1.1 Compiler-based approach

In compiler-based approaches, a conventional sequential language is used for initial specification without any modification. Then, a compiler parallelizes a sequential program automatically to find parallel regions from the application by itself. So, a key technique of the compiler-based approach is how to figure out parallel regions and identify data dependencies in the region from the manually described sequential program. Then, from the identified parallel regions, partitioner/mapper transforms each parallel region into a set of concurrent tasks, and maps them onto processors (or cores).

MAPS (MPSoC Application Programming Studio) [17] is a compiler-based MPSoC programming framework. It aims to generate multithreaded C code running on a predefined heterogeneous multiprocessor platform from a sequential code. Based on code analysis and profiling techniques, it performs automated code

partitioning as well as optimized spatial and temporal task to processor mapping [18]. The mapping is verified and refined with a simulation environment called MAPS Virtual Platform (MVP). Finally, the code generator generates the C codes for the respective cores from the partitioned graph.

Intel® compiler [19] is one of widely-used compilers. It analyzes the dataflow in loops to determine which loops can be safely and efficiently executed in parallel. The compiler generates executable code that divides the iterations as evenly as possible among the threads at runtime.

CriticalBlue Multicore Cascade [20] is a commercial tool-chain which synthesizes coprocessors to execute the parallel regions. It analyzes an application binary and constructs concurrent tasks from the application, and then it generates Cascade coprocessors and allocates the concurrent tasks onto the generated coprocessors. To achieve high performance, it considers pipelining, communication overhead, and cache configuration.

The key advantage of the compiler-based approach is that no burden of parallelization is imposed on programmers. However, it is not easy for a compiler to identify complex parallelism by itself. So the compiler-based approach can only effectively analyze parallel regions with a relatively simple structure.

2.1.2 Language-based approach

Since it is difficult for a compiler to identify parallel regions and extract various types of parallelism from the sequential program by itself, a language-based approach is widely adopted and used in industry and academia. In the language-based approach, a programmer specifies parallelism information, where

and how to parallelize the code, with supported annotations or APIs. Therefore, compilers in language-based approaches just need to focus on exploiting the specified parallelism according to the target platform. It can be categorized with follow two approaches: annotation-based and API-based extensions.

2.1.2.1 Language extension with annotations

OpenMP [21][22] is one of widely used annotation-based approaches. In OpenMP, a programmer specifies parallel region explicitly using the supported directives. Fork-join is the main model of parallel execution in the OpenMP framework. Figure 2-1 shows simple usage of directives in the OpenMP framework.

```
A(); B();  
#pragma omp parallel for  
for (i = 0; i < 99; i++)  
    C();  
D();
```

Figure 2-1. Usage of OpenMP directives

Even though OpenMP was originally developed for symmetric multiprocessor (SMP) computers with shared address-space, it is also used for heterogeneous multicore platforms such as and GPGPU [23] and IBM Cell [24][25] that cannot easily apply shared address-space based parallel programming. Similarly with OpenMP, StarSs [26] proposes a directive-based parallel programming model which supports various target platforms such as IBM Cell, GPU, SMP, and so on.

Also, Intel® Cilk Plus [27] supports simple language extensions to the C and C++ languages to express task and data parallelism. It provides three keywords which are surprisingly powerful model for parallel programming, while runtime and

template libraries offer a well-tuned environment for building parallel applications. Its runtime system operates on systems with hundreds of cores, so it can be easily adopted to many core systems.

The key benefit of this approach is that it relieves the compiler's burden of parallelism extraction while it gives only a little overhead to the programmer. However, target-specific optimization is not easy to be achieved by a translator only, since the annotation specifies the high-level information of the parallel region.

2.1.2.2 Language extension with APIs

Message passing interface (MPI) [28][29] is a standard specification for parallel programming, based on the message-passing model where each processor has private memory and communicates with other processors via message passing. It defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs. Also, MATLAB supports MPI-based parallel programming [30].

```
if(Rank==0) then {  
    A(); B();  
    for i=1 to num_of_proc: MPI_Send(...);  
}  
else then {MPI_Recv(...);}   
C();  
if(Rank==0) then {  
    for i=1 to num_of_proc: MPI_Recv(...);  
    D();  
}  
else then {MPI_Send(...);}
```

Figure 2-2. Usage of MPI APIs

Compute Unified Device Architecture (CUDA) [31][32] from NVIDIA is a parallel programming solution for their own GPUs. It defines a kernel, specified by compiler directives, as a function that will run on the stream processors in GPU. It also provides vendor-specific APIs to load the kernels into the GPU memory and to execute them. To define a programming standard that supports heterogeneous multicore platforms, Khronos announced an open standard for heterogeneous parallel programming, called OpenCL [33][34]. OpenCL is a language extension for data-parallel processing and defines a set of APIs, some of which are shown in Figure 2-3.

```
A(); B();  
kernel = clCreateKernel(progC, "C");  
work_size[0] = 99;  
range = clCreateNDRangeContainer(work_size, ...);  
clExecuteKernel(..., kernel, ..., range, ...);  
D();
```

Figure 2-3. Usage of OpenCL APIs

Compared with the annotation-based language extension, the APIs allow more low-level control of parallelism to the programmer. So the API-based language extension becomes an industrial standard for parallel programming since the application behavior can be easily specified at a low-level by using rich and well-defined APIs. However, it still depends on manual programming except parallel regions. So, as the complexity of the software increases, it is hard to implement and debug the program. Also, in the language- based approach, resource or timing constraints which are commonly given in modern embedded systems cannot be guaranteed.

2.1.3 Model-based approach

How to guarantee the given resource and real-time constraints in embedded systems becomes more important, a model-based approach is widely used for parallel programming. Because, in the model-based approach, an application is specified based on an abstracted model that simplifies the application behavior. Also potential parallelisms in the application can be explicitly exposed just by specification. Table 2-1 shows well-known models in the model-based approach.

Table 2-1. Category of widely-used models

Category	Models
State-oriented	Finite State Machine, Statechart [36], ...
Dataflow	Synchronous dataflow [4], Cyclo-static dataflow [75], ...
Process network	Kahn process networks [37], YAPI [40], ...
Heterogeneous	Ptolemy [35], PeaCE [11], ...

Especially, models of computation (MoCs) [60] which provide means for abstract and formal representation of the computational behavior are used for parallel programming. Kahn process network [37] and Synchronous dataflow [4] models are representative MoCs which are used in various model-based frameworks.

Distributed operation layer (DOL) [61] is a KPN-based design framework which enables the automatic mapping of applications onto the multiprocessor platform. It defines a set of computation and communication routines that enable the programming of distributed, parallel applications for the multiprocessor platform. These routines are subject to further refinement in the hardware dependent software (HdS) layer [62]. Also, it supports a mapping optimization technique [63] to

compute a set of optimal mappings of an application onto the multiprocessor platform, as well as parallel code generation [87] for multiprocessor platforms.

To extend the DOL framework, DAL (Distributed Application Layer) [59] is proposed which supports a software development framework for the model-driven development of multi- and many-core platforms. It combines an FSM model with the KPN model to express system-level behavior. It supports the design, the optimization, and the simultaneous execution of multiple dynamically interacting streaming applications on heterogeneous platforms. Also it supports automatic code generation for XeonPhi [88], Intel SCC [89], and distributed linux [90] platforms.

Daedalus [41][42] is a system-level design flow for the design of MPSoC based embedded multimedia systems. It also supports a KPN-based specification using PNgen compiler [43] and offers a fully integrated tool-flow in which design space exploration (DSE), system-level synthesis, application mapping, and system prototyping of MPSoCs are highly automated.

Even though the KPN model is widely used in model-based design frameworks for streaming applications because of its expressiveness, it has limitations of static analyzability especially for timing constraints. In contrary to the KPN model, the

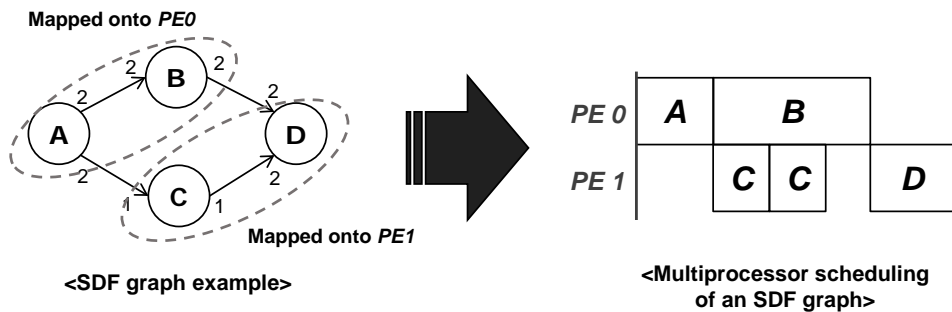


Figure 2-4. An SDF graph and schedule example

SDF model has desirable properties especially for static analyzability. Because, as shown in Figure 2-4, a schedule can be constructed in compile-time, we can find static schedules which guarantee the given real-time/resource constraints [44] [45].

StreamIt [46] is a programming language and a compilation infrastructure, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient and effective mapping to a wide variety of target architectures, including multicore architectures, and clusters of workstations. The StreamIt language has some extensions to the SDF model such as the split-join mechanism that may express data parallelism. Also, it supports the StreamIt compiler [84] which performs fully automatic load balancing, graph layout, communication scheduling, and routing [85][86] as well as generates target executable parallel code from the StreamIt specification.

Daedalus RT [47] is an extension of Daedalus framework for hard-real time streaming applications. It automatically derives their equivalent CSDF [75] graphs from the PPN [76] models. The derived CSDF graphs are used in such analysis which applies hard-real-time multiprocessor scheduling theory to schedule the applications in a way that temporal isolation and a given throughput of each application are guaranteed [77].

Gedae [48] is a software development tool based on a dynamic data-flow (DDF) [38] model. Through data-flow analysis, it determines the execution order, queue-sizes, memory layout, and so on. Also, it provides a simulation environment that is based on a multiprocessor virtual machine, which enables it to detect memory access violations. Gedae Idea Compiler [91] is supported which automates the implementation of processor specific code required to run algorithms on today's multicore and complex memory architectures.

PREESM [49] is an SDF model-based design framework which simulates signal processing applications and generates code for heterogeneous multi/many-core embedded systems. Inputs of the PREESM tool are an algorithm graph, an architecture graph, and a scenario which is a set of parameters and constraints that specify the conditions under which the deployment will run. It support a parameterized and hierarchical extension of Synchronous Dataflow (SDF) graphs named PiSDF [92]. The architecture graph is named System-Level Architecture Model (S-LAM) [93]. From these inputs, PREESM maps and schedules automatically the code over the multiple processing elements [94] and generates multi-core code [95].

However, the SDF model has a typical weakness of expression capability, especially for dynamic behavior specification. First, it mainly focuses on application-level specification such as streaming applications. So system-level specification is not considered. Next, it has restricted execution semantics such as fixed sample rates. So it cannot express applications which consist of multiple behaviors depending on its algorithm or multiple implementations depending on QoS. To overcome such limitations, there exist several extensions based on the KPN model and the SDF model. They will be introduced in the next chapter.

2.2 HOPES framework

The proposed approach is implemented on the HOPES framework [15]. HOPES framework is a model-based design framework which supports a parallel programming environment for non-trivial heterogeneous multiprocessors with various design constraints on hardware cost, power, real-time performance. Unlike other HW/SW codesign environments, HOPES framework puts more emphasis on the implementation of software components, while other environments focus on the codesign of hardware and software modules that includes HW/SW partitioning, cosynthesis, and cosimulation, as well as take little account of multiprocessor architecture that heavily affects the parallel execution of software.

As shown in Figure 2-5, HOPES framework supports the overall design steps: system specification, performance estimation, design space exploration, and automatic code generation. Starting from a target-independent behavior specification and a given set of candidate hardware architectures and available processing elements, HOPES framework can explore the design space to find an

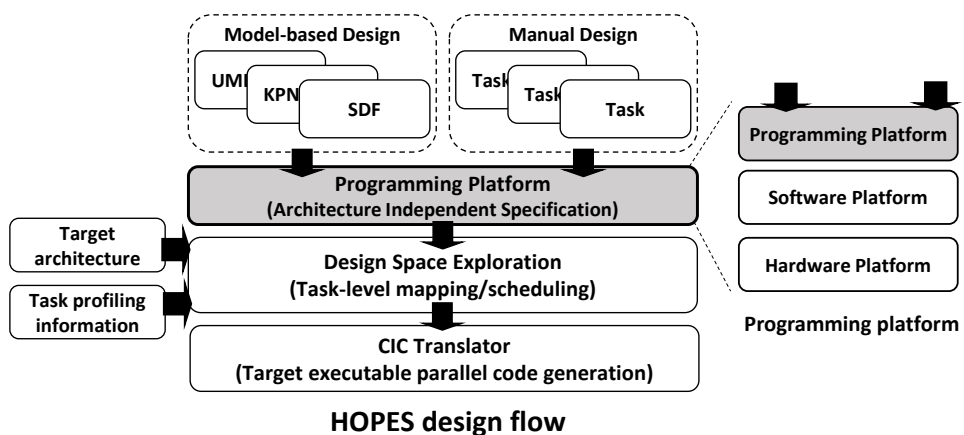


Figure 2-5. HOPES design flow

optimal system configuration and mapping of applications, and synthesize the software and hardware components in a unified framework.

Especially, HOPES framework introduces a new notion of programming platform called CIC (Common Intermediate Code) that hides the underlying software and hardware platform from the application programmer. As the name implies, the CIC model is not defined as a front-end specification model, but an intermediate specification model, meaning that HOPES framework can accommodate various front-end specification models as long as the front-end specification model can be translated into the CIC model. In fact, the CIC model can be understood as an execution model of tasks at the OS (operating system) level. At the OS level, the system behavior is represented as a set of tasks no matter what the front-end specification model is. As shown in Figure 2-5, communication and synchronization between tasks and scheduling of tasks are heavily dependent upon the underlying software platform and hardware platform. So, the CIC model defines the execution model of tasks at the OS level and enforce the system to keep the semantics of the execution model, then it will be able to run on any hardware and software platform since the execution model is defined as platform-independent.

Because we extend the CIC model to support dynamic behavior specification, before we introduce the extended features, details of the base CIC model will be explained in the next chapter.

2.3 Common Intermediate Code (CIC) Model

As previously mentioned, the proposed task model is extended based on the CIC task model [15] that an operating system (OS) handles, based on the observation that all applications on hardware platforms share a common execution model at the OS level regardless of what model of computation is assumed for a front-end specification: an application consists of threads or tasks that are scheduled by an operating system. It defines defining how to construct the tasks, when to execute the tasks, and how to perform task synchronization and inter-task communication. Also, it adopts a hierarchical composition of different models of computation to express the system behavior at two different levels. At the top level, the CIC model expresses the system behavior with a process network. If an application can be specified by an SDF graph, the application is encapsulated as a super node that contains the SDF graph at the bottom-level.

The top-level process network consists of CIC tasks and channels as depicted in Figure 2-6. There are two types of CIC tasks depending on the triggering condition of tasks: *time-driven* and *data-driven*. A time-driven task is triggered when the specified time is reached. The input channels of a time-driven task are single-entry buffers that store the most recent data samples. An I/O task that interfaces with the outside is usually designated as a time-driven task. On the other hand, a data-driven

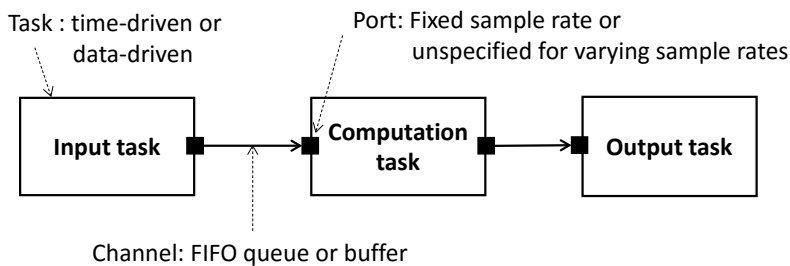


Figure 2-6. CIC task graph

task is triggered by the arrival of data samples on the input ports. The input channels of a data-driven tasks are assumed to be FIFO queues. A data-driven task basically follows the semantics of the KPN model that performs blocking read, non-blocking write access to the channels.

Definition 3.1 (Top-level task graph). In the CIC model, a top-level task graph is described by a tuple (T, C, D) , where

- $T = \{t_i | i = 0, \dots, \# \text{ of tasks}\}$ is the finite set of computational tasks.
 - For each task $t \in T$, $drivenType(t) \in \{time\ driven, data\ driven\}$ is the driven type of task t .
 - For each task $t \in T$, $Port(t) = inPort(t) \cup outPort(t)$ is the set of ports to send/receive data with other tasks where $inPort(t)$ is a set of input ports and $outPort(t)$ is a set of output ports.
- $C = \{c_i | i = 0, \dots, \# \text{ of channels}\}$ is the finite set of channels.
 - For $\forall c \in C$, $channelType(c) \in \{Message\ queue, Buffer\}$ is the type of channel c . Each port $p \in Port(t)$ is uniquely connected to one channel c and every channel to two ports.
- $D = \{d_i | i = 0, \dots, \# \text{ of channels}\}$ is the finite set of the number of initial tokens in each channel. $d_i \in \{0\} \cup \mathbb{N}$ denotes the number of initially stored tokens in channel c_i .

If $hasSubGraph(t) = False$, behavior of task t is specified with three functions, TASK_INIT, TASK_GO, and TASK_WRAPUP in Figure 2-7. As the name implies, the TASK_INIT function is executed when the task is initialized and the TASK_WRAPUP function is executed just before it is terminated. The TASK_GO function is the main body that will repeat until the task is terminated.

A CIC task accesses a channel with target-independent generic APIs, MQ_SEND/RECEIVE and BUF_SEND/RECEIVE. The MQ_RECEIVE API performs blocking read operation to the associated input port while the MQ_SEND API performs non-blocking write operation to the associated output port. And the BUF_SEND/RECEIVE API performs non-blocking read/write operations to the associated port. Since the CIC model is defined at the OS level, the CIC task assumes that there is a supervisor that schedules the CIC tasks and provides supervisory services to the CIC tasks. Thus we define another generic API, SYS_REQ, that requests a service to the supervisor. The first argument of the SYS_REQ API defines the service command whose list will be shown later. In principle, a CIC task does not use platform-specific APIs for portability. The generic APIs will be translated into target-specific APIs at the code generation step. We may define a CIC task that uses platform-specific APIs for efficient implementation at the expense of portability.

```
TASK_INIT{ /* task initialization code */ };
TASK_GO {
    /* generic API for data read from an input port */
    MQ_RECEIVE(port_name, data, size);
    ...
    /* generic API for system service request */
    SYS_REQ(command, argument_list);
    ...
    /* generic API for data write to an output port */
    MQ_SEND(port_name, data, size);
}
TASK_WRAPUP { /* task wrapup code */ };
```

Figure 2-7. CIC task code template

The number of data samples consumed or produced per execution of a task can be specified explicitly for each input or output port. The sample rate is specified, if it is fixed and not changing at run time. Otherwise, the sample rate is assumed to be varying at run time. If the input sample rates of all input ports are specified, the data-driven task becomes an SDF task that follows the execution semantics of the SDF model. If all tasks in a CIC subgraph are SDF tasks, the CIC subgraph becomes an SDF subgraph. Since the SDF model has many merits from static analyzability, it is highly recommended to identify SDF subgraphs as much as possible at the top level until no more SDF subgraph can be identified. And each subgraph is replaced by a super node at the top-level to make it a two-level hierarchical graph.

Definition 3.2 (Hierarchical composition). In a top-level task graph,

- For each task $t \in T$, $hasSubGraph(t) \in \{True, False\}$ denotes whether a task t has a subgraph. If $hasSubGraph(t) = True$, $SubGraph(t)$ is an SDF subgraph of computation task t .

Definition 3.3 (Second-level task graph). In the CIC model, a second-level task graph is described by a tuple (T, C, D) , where

- $T = \{t_i | i = 0, \dots, \# \text{ of tasks} \}$ is the finite set of computational tasks.
 - For each task $t \in T$, $drivenType(t) = data \ driven$ is the driven type of a task t .
 - For each port $p \in Port(t)$, it has a fixed sample rate $Rate(p) \in \mathbb{N}$.
- $C = \{c_i | i = 0, \dots, \# \text{ of channels} \}$ is the finite set of channels.
 - For $\forall c \in C$, $channelType(c) = Message \ queue$ is the type of channel c . Each port $p \in Port(t)$ is uniquely connected to one channel c and every channel to two ports.

- $D = \{d_i | i = 0, \dots, \# \text{ of channels}\}$ is the finite set of the number of initial tokens in each channel. $d_i \in \{0\} \cup \mathbb{N}$ denotes the number of initially stored tokens in channel c_i .

In the top-level task graph, if $hasSubGraph(t) = True$, each port $p \in Port(t)$ should be mapped to a port in the subgraph $SubGraph(t)$. Definition 3.4 shows a port mapping relation between internal/external ports of the task.

Definition 3.4 (Port mapping between internal/external ports).

- For each task $t \in T$ in the top-level graph, if $hasSubGraph(t) = True$
 - Each port $p \in inPort(t)$, $PortMap(p) = ip$ where ip is one of input ports in the subgraph $SubGraph(t)$.
 - Each port $p \in outPort(t)$, $PortMap(p) = op$ where op is one of output ports in the subgraph $SubGraph(t)$.
 - Each port $p \in Port(t)$ is uniquely mapped to one port in the subgraph $SubGraph(t)$.

Because the CIC model is based on process network and dataflow models, it has limitations of expression capability and static analyzability. Therefore, we extend the CIC task model to support dynamic behavior specification in both system-level and application-level based on formal models such as dataflow and finite state machine. Details will be explained in the next chapter.

Chapter 3 Dynamic Behavior Specification

3.1 Problem definition

Before we explain how the proposed approach supports dynamic behavior specification for both system-level and application-level, we need to clarify and define the dynamic behavior which will be covered in this dissertation.

3.1.1 System-level dynamic behavior

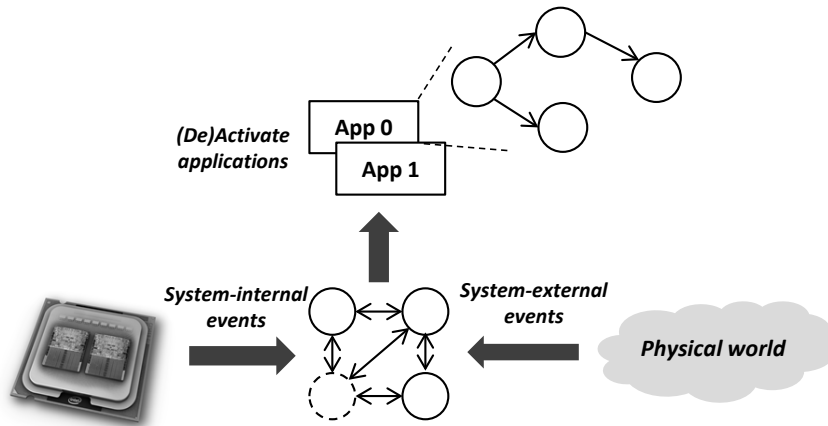


Figure 3-1. System-level dynamic behavior

In the system level, the system reacts system internal/external events, so a behavior of the system will be changed depending on those events. We suppose that the system-level behavior can be classified to a number of use-cases. In each use-case, which application tasks are activated and functional behavior of each

application task will be given and fixed. Also, each application task may has a real-time constraint such as latency and throughput for each use-case. We assume that the number of use-cases in the system is finite, then we can capture system-level dynamic behavior with the finite number of use-cases and fixed behavior in each use-case. So it can be expressed by a finite state machine and dataflow graphs.

3.1.2 Application-level dynamic behavior

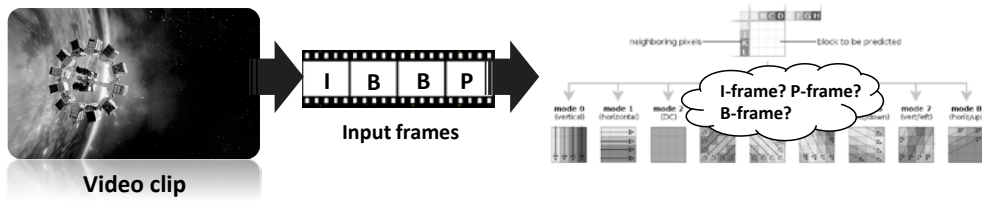


Figure 3-2. Application-level dynamic behavior

In the application level, as the complexity of modern embedded applications increases, an application may consists of a set of conditionally invoked functions. As shown in Figure 3-2, a video CODEC algorithm such as H.264/MPEG-4 [96] is a representative example which behaves differently depending on the type of encoded input frame: I, P, or B type. Also an application can have multiple implementations of the same algorithm depending on QoS. Therefore, we assume that an application can have multiple modes (or behaviors) depending on input data or system internal/external events, and the number of modes is finite. Then we can capture application-level dynamic behavior with the finite number of modes and fixed behavior in each mode.

In summary, we consider the finite number of dynamic behaviors in both system-level and application-level. It allows us to perform the compiler-time analysis for each use-case and each mode (or behavior) to guarantee the given resource and real-time constraints. Therefore, it needs to support dynamic behavior specification for the finite number of dynamic behaviors in both system-level and application-level using formal models.

3.2 Related work

While there are numerous specification methods proposed to express the dynamic behavior of a system, we review some representative ones that are based on the SDF model or the KPN model since they are closely related with the proposed specification methods.

There are two popular approaches to express the application-level dynamism. One approach is to define an extended model of the based dataflow model. Dynamic dataflow (DDF) [38] and Boolean dataflow (BDF) [39] are two examples of extended SDF models to express the non-deterministic behavior. PSDF (Parameterized Synchronous Dataflow) [8][9] uses a meta-modeling technique for run-time adaptation of parameters in a structured way. In PSDF, the dynamic behavior of a task is modeled by parameters and the parameters can be changed at run-time after each iteration of the schedule is completed. For the KPN model, YAPI [40] adds the probing capability by introducing the select statement to check the existence of the input data sample. Then a task can read a control input event asynchronously to change its internal behavior at run-time.

The other approach is to combine the dataflow model together with some form of reactive behavior. The *-chart [50] introduces dynamism through a finite state

machine that executes an iteration of an SDF graph in each state. The model combination is hierarchical; each state has an SDF graph inside and states may have different SDF graphs. FunState [51], proposed as an internal design representation for codesign process, separates dataflow and control. Activation of tasks is controlled by a finite-state machine, similar to the semantics of activity charts in statecharts implementations [52]. The SystemoC [53] was introduced as a limitation of the FunState model to integrate FSMs with dataflow models. Each actor is associated with an actor FSM that controls the communication behavior as well as action functions. The FSM-based SADF [54], shortly FSM-SADF, is a restricted form of the general SADF model [55][56] that specifies each mode of operation, or a scenario, with an SDF graph. To specify multiple scenarios and their transition, it defines a special control task, called detector that has an FSM inside. The detector task sends the control information to the normal computation tasks that may change its behavior. Our proposed method of intra-application dynamism is basically equivalent to the FSM-SADF model except for minor differences in syntax and mode change mechanism. Instead of combining a dataflow model with a FSM, RPN (Reactive Process Network) [57][58] is a single unified model that regards reconfiguration as an event handling. In RPN, a process network is hierarchically composed; a process may contain an RPN inside. The dynamic behavior is specified by separating event inputs and data inputs and defining how configurations are changed according to input events.

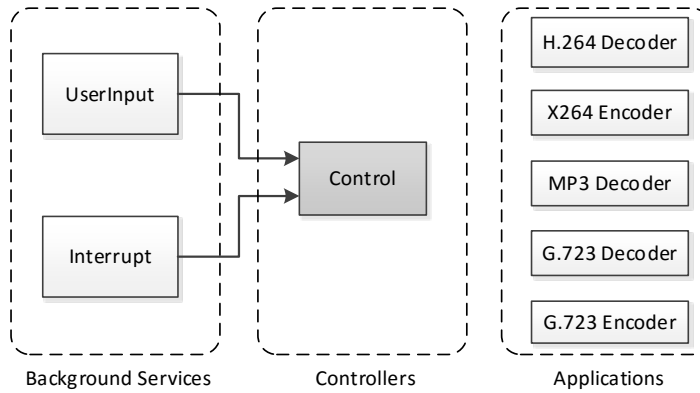
There are also several ways to specify system-level dynamism. FunState and RPN can also be used for this purpose since they allow hierarchical composition of the components. As previously mentioned, DAL (Distributed Application Layer) [59] is an extended model of DOL (Distributed Operation Layer) [61] where each application is specified by a KPN. In DAL, an FSM specifies all use cases and how

transition between use cases occurs. Note that the number of states corresponds to the number of use cases which may explode as the number of applications increases. In STATEMATE [52], the statechart as a central FSM describes the entire system behavior while task models do not possess formal semantics. Since the FSM model controls task executions, complicated task scheduling should be described manually. Also, Ptolemy [35] is a pioneering research framework that supports heterogeneous modeling in a hierarchical fashion. It supports modeling, simulation, and design of concurrent, real-time, embedded systems with allowing heterogeneous mixtures of models of computation. Our proposed method is inherited from our earlier work, PeaCE [11][12]. In PeaCE, a control task whose internal behavior is specified by a finite state machine is distinguished from computation tasks. The control task plays the role of supervisory task that controls the execution status of applications that are specified by a dataflow model.

Recently, it is argued that timing should be included as a part of behavior specification since timing correctness is as important as value correctness in system functionality. Thus PTIDES (Programming Temporally Integrated Distributed Embedded System) [13][14] was recently proposed. PTIDES is a programming model based on discrete event model of computation for distributed real-time embedded system. Inspired by this work, we specify the timing requirements in the control task specification.

To our best knowledge, the proposed specification is unique in that we use different specification methods for application-level and system-level dynamism while each application behavior is specified with the SDF model. By utilizing the static analyzability of the SDF model, we are able to perform hybrid mapping of tasks to support dynamism, satisfying the real-time constraints.

3.3 Motivational example



Categories	Tasks	Description
Input Tasks	<i>UserInput</i>	Sensor task for sensing events from a user
	<i>Interrupt</i>	Receiver task for detecting phone call signal
Control Tasks	<i>Control</i>	Control task to control all applications
Application Tasks	<i>H.264 Decoder</i>	H.264 decoder task for video play/phone
	<i>X264 Encoder</i>	x264 encoder task for video phone
	<i>MP3 Decoder</i>	MP3 decoder task for music play
	<i>G.723 Encoder</i>	G.723 encoder task for video phone
	<i>G.723 Decoder</i>	G.723 decoder task for video phone

Figure 3-3. A multi-mode multimedia terminal example

As a motivational example, a multi-mode multimedia terminal system [64] is considered as shown in Figure 3-3. The system has two input tasks running in the background; one, denoted by *UserInput*, is for receiving the user command, and the other, denoted by *Interrupt* for detecting the incoming phone. It has five applications: *H.264 Decoder*, *x264 encoder*, *MP3 decoder*, *G.723 encoder*, and *G.723 decoder*. There are four use cases as shown in Table 3-1.

Table 3-1. Use cases of the multi-mode multimedia terminal system in Figure 3-3

Use case	Active applications
<i>Standby</i>	-
<i>VideoPhone</i>	G.723 Decoder, G.723 Encoder, H.264 Decoder, x264 Encoder
<i>VideoPlay</i>	MP3 Decoder, H.264 Decoder
<i>MusicPlay</i>	MP3 Decoder

In the Standby mode, no application task is running and the system waits until it receives a user input or a receiving phone call. Depending on the user input, the system changes the mode of operation and activates the applications for the corresponding use case. When a phone call is received during the other modes, the system suspends the currently active applications and switches to the *VideoPhone* mode. After the call is completed, it resumes the suspended applications and goes back to the stand-by mode when all the applications terminate.

In the proposed specification model, we define a control task as shown in Figure 3-3. The control task controls the execution status of applications and performs state transitions based on the current state and input data from the *UserInput* task or *Interrupt* task. The next chapter will explain the control task in details.

In this example, an application is specified by a single CIC task at the top-level. An application can be specified by an SDF graph inside the CIC task. As explained earlier, the granularity of a task should be determined considering the trade-off between scheduling overhead and performance gain by parallel execution. The *G.723 decoder* and *G.723 encoder* are specified by a single CIC task respectively while the *H.264 decoder*, *x264 encoder*, and *MP3 decoder* applications need to be specified by a dataflow graph inside each CIC task. Among them, *H.264 decoder* has intra-application dynamism since the behavior varies depending on the type of

input video frame, *I-frame* or *P-frame*. Since the SDF graph cannot express such dynamic behavior, we extend the base CIC model to use a hybrid model of the SDF model and the FSM model inside the CIC task.

3.4 Control task specification for system-level dynamism

To model the reactive behavior of the system, we define a special type of a CIC task, called CIC control task, whose internal behavior is specified by an FSM while it behaves as a CIC task from the outside. So, the entire system can be specified by computational tasks (T), control tasks (CT), and channels for communication between tasks. A task named as *Control* in Figure 3-3 becomes a CIC control task. The definition of the top-level task graph in Definition 3.1 should be extended to consider the CIC control task.

Definition 4.1 (Control task in the top-level task graph). In the CIC model, a top-level task graph is described by a tuple (T, CT, C, D) , where

- $CT \subseteq \{ct_0\}$ is a set of control tasks. Only one control task can be specified in the top-level task graph.
- The internal behavior of the control task is specified with an FSM \mathcal{F} .

3.4.1 Internal specification

Basically, the internal behavior of the CIC control task is specified with FSMs and it allows hierarchical and concurrent compositions of FSMs with AND/OR composition [36]. However, for simple coordination, we assume that the internal behavior of the CIC control task is given by a single FSM through AND/OR decomposition although it can be specified with hierarchical and concurrent FSMs. Also, similarly with fFSM [97], the CIC control task can have variable states that

can be regarded as a separate concurrent FSM graph in which each state is mapped to a value that the variable state can have. Then, it can be formulated as follow:

Definition 4.2 (Definition of a finite state machine). An FSM $\mathcal{F} = (S, R, E, V, T, s_o, \sigma, \gamma, \varepsilon, \rho, \mu)$ consists of

- The set of states S , the set of transitions R , the set of events E , the set of variables V , the set of application tasks T , an initial state s_o , and five mapping functions: $\sigma, \gamma, \varepsilon, \rho$, and μ .

First, the overall use-cases of the system are specified with the finite number of states in which each state corresponds to each use-case of the system.

Definition 4.3 (Definitions of states).

- $S = \{s_i \mid i = 1, \dots, \# \text{ of states} \}$ is a finite set of states.
- $V \subset S$ is a finite set of variable states.
- $s_o \in S$ is an initial state. An initial state should exist only one for the FSM.

In the CIC model, we have three different event types: internal event, external event, and timeout event. Internal event (IE) is an event when the execution status of a computation task changes. External event (OE) is an event which is received from input ports. And timeout event (TE) is an event which is generated when the set time is passed. Definition 4.4 shows the definition of event sets.

Definition 4.4 (Definitions of events).

- $IE = \{(i_{n_k}, i_{c_k}) \mid k = 1, \dots, \# \text{ of internal events} \}$ is a finite set of internal event names and of the corresponding finite set of conditions. Each condition i_{c_k} consists of a finite set of tuples $(t, Status)$ where $t \in T$ and $Status = Run/Stop/Suspend$.
- $OE = \{(e_{n_k}, e_{c_k}) \mid k = 1, \dots, \# \text{ of external events} \}$ is a finite set of external event names and of the corresponding finite set of conditions. Each condition e_{c_k} consists of a finite set of tuples (p, c, v) where $p \in$

$InPort(ct_0)$, $c \in \{<, >, \leq, \geq, =, \neq\}$, and v denotes a value.

- $TE = \{(t_{n_k}, t_{c_k}) \mid k = 1, \dots, \# \text{ of timeout events}\}$ is a finite set of timeout event names and of the corresponding finite set of conditions. Each condition $t \in t_{c_k}$ consists of a finite set of tuples (v, u) where v denotes a value, and u denotes a unit of time.
- $E = IE \times OE \times TE$ is a finite set of events which consist a combination of three types of event.

Also, a transition connects two different states: one is a source state and the other is a destination state. It is also associated with a condition which is specified with a boolean expression composed of a set of events and variable states.

Definition 4.5 (Definitions of transitions).

- $R = \{r_k \mid k = 1, \dots, \# \text{ of external events}\}$ is a finite set of transitions. Each condition $r \in R$ consists of a finite set of tuples (s_i, f, s_j) where $s_i \in S$ and $s_j \in S$ are a source state and a destination state, respectively. And function f is a boolean function, such that $f : OE \cup IE \cup TE \cup V \rightarrow \{True, False\}$

In each state $s \in S$, which applications are (de)activated or suspended is given as well as each application may have multiple modes and timing requirements that can be controlled by the control task. So, Definition 4.6 shows the definition of applications in the CIC model.

Definition 4.6 (Definitions of task properties). Each computation task $t \in T$ has follow four properties:

- $Status(t) = Run/Stop/Suspend$ denotes the execution status of computation task t .
- $Mode(t) = \{m_k \mid k = 1, \dots, \# \text{ of modes}\}$ is a finite set of modes of computation task t . It can be controlled by the control task as well as by itself for application-level dynamism. It corresponds to the modes in the MTM which will be explained in Chapter 3.5, in detail.

- $Param(t) = \{(p_{n_k}, p_{v_k}) \mid k = 1, \dots, \# \text{ of parameters}\}$ is a finite set of parameter names and of the corresponding values.
- $Const(t) = \{(c_{t_k}, c_{v_k}) \mid k = 1, \dots, \# \text{ of timing constraints}\}$ is a finite set of constraint types and of the corresponding values. The constraint type c_{t_k} is either *Throughput* or *Latency*.

Next, we need to define four mapping functions: $\sigma, \gamma, \varepsilon, \rho$, and μ . They can be defined as follow:

Definition 4.7 (Definitions of mapping functions).

- $\sigma : S \rightarrow T$ is a mapping function which assigns each state $s \in S$ to a set of active tasks $\sigma(s) \subseteq T$
- $\gamma : S \rightarrow T$ is a mapping function which assigns each state $s \in S$ to a set of suspended tasks $\gamma(s) \subseteq T$.
- $\varepsilon : S \rightarrow Mode(t)$ is a mapping function which assigns each state $s \in S$ to a set of modes $\varepsilon(t, s) \in Mode(t)$ for active tasks $\sigma(s)$.
- $\rho : S \rightarrow Param(t)$ is a mapping function which assigns each state $s \in S$ to a set of parameters $\rho(t, s) \in Param(t)$ for active tasks $\sigma(s)$.
- $\mu : S \rightarrow Const(t)$ is a mapping function which assigns each state $s \in S$ to a set of timing constraints $\mu(t, s) \in Const(t)$ for active tasks $\sigma(s)$.

In summary, each use-case of the system is mapped onto each state s in the control task. In each state $s \in S$, active task set $\sigma(s) = \{t_0, t_1, \dots\}$ and suspended task set $\gamma(s) = \{t_0, t_1, \dots\}$ are given, and $\varepsilon(t, s)$, $\rho(t, s)$ and $\mu(t, s)$ are given for the functional behavior and the real-time constraint of each task $t \in \sigma(s)$. Then we can determine the behavior of the system for each use-case, and it enables us to perform the compile-time analysis to guarantee the given resource and real-time constraints to the system. Details will be explained in Chapter 4.

The internal specification of the *Control* task for the MMT example is shown in Figure 3-4. There are four states where each state corresponds to a use case. The initial state is the *Standby* state, denoted as a bold circle in the figure. The *VideoPhone* state can be entered from any other state, because a phone call can arrive at any time. At the right-side in Figure 3-4, we show which applications are active in each use case. Also, once a triggering event is received, the state transition occurs based on the received events and the variable states (*V*) that a CIC control task maintains. In the example of Figure 3-3, we need to store the previous state as a variable state when a *phone_call* event is received. After the phone conversation is completed, we have to return to the previous state. So, $V = \{prevState\}$. The FSM of Figure 3-4 shows all state transitions for the example of Figure 3-3.

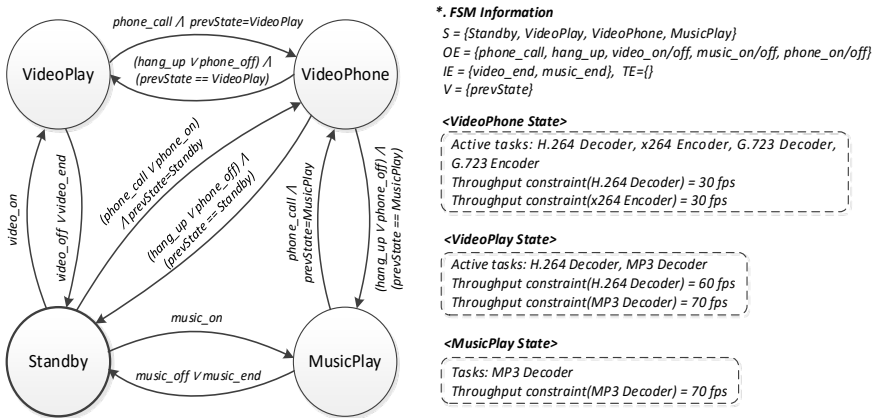


Figure 3-4. FSM in the control task in Figure 3-3

3.4.2 Action scripts

To define control interaction between a control task and computational tasks, we use system APIs that request supervisory services. It is similar to the use of action scripts of the statechart [52] in STATEMATE. In Table 3-2, it shows the list of control APIs where the first two categories belong to action scripts.

Table 3-2. Control APIs in the proposed model

Category	APIs	Description
<i>Execution</i> <i>Status</i> <i>Control</i>	<code>SYS_REQ(RUN_TASK, task_name);</code>	Run the task
	<code>SYS_REQ(STOP_TASK, task_name);</code>	Terminate the task
	<code>SYS_REQ(SUSPEND_TASK, task_name);</code>	Suspend the task
	<code>SYS_REQ(RESUME_TASK, task_name);</code>	Resume the task
	<code>status=SYS_REQ(CHECK_TASK_STATE, task_name);</code>	Check the task status
<i>Behavior</i> <i>Control</i>	<code>SYS_REQ(CHANGE_MODE, task_name, mode_name);</code>	Change a mode
	<code>p_value = SYS_REQ(GET_PARAM_INT/FLOAT, task_name, param_name);</code>	Get the value of a task parameter
	<code>SYS_REQ(SET_PARAM_INT/FLOAT, task_name, param_name);</code>	Change a value of a task parameter
<i>Timing</i> <i>Control</i>	<code>time_base = SYS_REQ(GET_CURRENT_TIME_BASE);</code>	Get current time
	<code>timer_id = SYS_REQ(SET_TIMER, time_base, offset);</code>	Set timer with an offset
	<code>ret = SYS_REQ(GET_TIMER_ALARMED, timer_id);</code>	Check the timer
	<code>SYS_REQ(RESET_TIMER, timer_id);</code>	Initialize the timer
<i>Real-time constraint</i>	<code>SYS_REQ(SET_THROUGHPUT/LATENCY, task_name, val, unit);</code>	Set throughput/latency requirement of the task

3.4.2.1 Control task triggering specification

A CIC control task is basically triggered by an event. As defined in Definition 4.4, there are three kinds of events that trigger a control task: $E = OE \times IE \times TE$

- OE denotes the set of external events. An external event is received from an input port of a CIC control task. As shown in Figure 3-3, an input interface with the outside of the system is modeled as a source CIC task without any input port. In Figure 3-3, two tasks, *UserInput* and *Interrupt*, deliver external events to the CIC control task; that is $OE = \{phone_call, hang_up, video_on/off, music_on/off, phone_on/off\}$ where the first two events come from the *Interrupt* task and the remaining events come from the *UserInput* task. Figure 3-5 shows an example code for control task triggering by system external events.

```
TASK_GO{
    while(true){
        int size = MQ_AVAILABLE(port_user_input);
        if(size > sizeof(int)){
            MQ_RECEIVE(port_user_input, &input, sizeof(int));
            break;
        }
        ...
    }
}
```

Figure 3-5. Example code for control task triggering by system external events

- IE denotes the set of internal events. A CIC control task is also triggered by an event that the supervisor generates internally. The hidden supervisor monitors the execution status of an application and generates an internal event if the execution status changes. For instance, when an application terminates, an internal event is generated. In Figure 3-3, $IE = \{video_end, music_end\}$. Figure 3-6 shows an example code for control task triggering by system internal events.

```

TASK_GO{
    while(true){
        int state = SYS_REQ(CHECK_TASK_STATE, "H264Dec");
        if(state == TASK_STATUS_STOP)    break;
        ...
    }
}

```

Figure 3-6. Example code for control task triggering by system internal events

- *TE* denotes a set of timeout events. We assume that the system has timer modules in the system regardless of how they are implemented. A timer can be implemented as a hardware component or a software module. To set a timer in a state, we start a new timer whose timeout value is the time offset given as an argument; the return value defines the timer id. As explained above, a timeout event is generated if the system timer is advanced by the given time offset from the current time. Once a timer is set, we can check the time advancement of a timer with its timer id. Before the timer expires, we may reset the timer to avoid a timeout event from being generated. Figure 3-7 shows an example code for control task triggering by timeout events.

```

TASK_GO{
    while(true){
        if(timer_id == -1)
            int timer_id = SYS_REQ(SET_TIMER, 3, "sec");
        else{
            int ret = SYS_REQ(GET_TIMER_ALARMED, timer_id);
            if(ret == 0){
                SYS_REQ(RESET_TIMER, timer_id);
                break;
            }
        }
        ...
    }
}

```

Figure 3-7. Example code for control task triggering by timeout events

3.4.2.2 Execution status and behavior control specification

A CIC control task can control execution status of the task as well as its execution mode and parameter value to control functional behavior of the task.

- Execution status ($Status(t)$): As previously mentioned, in each state $s \in S$, an active task set $\sigma(s) = \{t_0, t_1, \dots\}$ and a suspended task set $\gamma(s) = \{t_0, t_1, \dots\}$ are given. For this, a CIC control task can control execution status of computation tasks using system request APIs as shown in Figure 3-8.

```
TASK_GO{
    switch(current_mode){
        case STATE_VIDEO_PHONE:
            SYS_REQ(SUSPEND_TASK, "H264Dec_VIDEO");
            SYS_REQ(RUN_TASK, "H264Dec_PHONE");
            ...
    }
```

Figure 3-8. Example code for execution status control

- Parameter ($Param(t)$): A CIC computation task can have parameters for its functional behavior. When a CIC control task sets a value of the parameter using system request APIs, a CIC computation task can get a changed value of the parameter and it can determine its functional behavior.

```
TASK_GO{
    switch(current_mode){
        case STATE_MUSIC_PLAY:
            SYS_REQ(SET_PARAM_INT, "MP3Dec", "volume", 30);
            ...
    }
```

Figure 3-9. Example code for task parameter control

- **Mode ($Mode(t)$):** A CIC computation task can have modes for its functional behavior. As previously mentioned, modes of the CIC computational task are used for multiple behaviors of the algorithm as well as different implementations of the same algorithm for different QoS. In the case of the latter, a CIC control task can set an execution mode of the computational task using system request APIs.

```

TASK_GO{
    switch(current_mode){
        case STATE_MUSIC_PLAY:
            SYS_REQ(CHANGE_MODE, "MP3Dec", "low-power");
            SYS_REQ(RUN_TASK, "MP3Dec");
            ...
    }
}

```

Figure 3-10. Example code for task mode control

3.4.2.3 Timing requirement control specification

In the proposed CIC model, we specify the timing requirements for the system inside the control task. It allows us to set up the timing requirement of applications; one is to set the throughput constraint and the other to set the latency constraint. The throughput requirements of applications are displayed at the right side of Figure 3-4 for each use case. The throughput constraint can be specified to a time-driven task only. On the other hand, the latency constraint can be specified to both a time-driven task and a data-driven task. The latency constraint means the deadline that the task should complete once released. A task is said released when the triggering condition of the task is satisfied.

```
TASK_GO{
    switch(current_mode){
        ...
        case STATE_VIDEO_PLAY:
            SYS_REQ(SET_THROUGHPUT, "H264Dec", 60, "fps");
            SYS_REQ(RUN_TASK, "H264Dec");
            ...
    }
}
```

Figure 3-11. Example code for timing requirement control

Note that the timing requirement for a CIC computational task may vary depending on the use cases. For example, the throughput requirement for the *H.264 decoder* task is different between *VideoPhone* and *VideoPlay* states; generally we need higher throughput for video play than for video phone conversation. We can change the timing requirement simply calling the corresponding system API in a state.

It should be emphasized that the timing requirement is a part of initial specification. Therefore, the timing requirements should be satisfied in the subsequent design steps; hardware platform decision, mapping and scheduling of applications onto the hardware platform, and target code generation.

3.5 MTM-SDF specification for application-level dynamism

To express the dynamic behavior of an application, we use a similar method as the FSM-based SADF assuming that the number of dynamic behaviors is finite. A dynamic behavior is called an operation mode of the application. Since the assumed task granularity is large in the CIC model, the number of modes of an application is not large. For example, we define only two modes for *H.264 decoder* in Figure 3-3.

3.5.1 MTM specification

While the base CIC model allows a CIC task to have an SDF subgraph, in the extended CIC model, a CIC task has an MTM (Mode Transition Machine)-SDF model. In an MTM-SDF model, a node may have multiple behaviors depending on the mode of operation, called scenario. In each mode, the overall graph becomes an SDF graph. The MTM is a tabular specification of an FSM that describes the mode transition rules for the task graph. It is the same as the FSM-based SADF except for the syntax of MTM. But there are some minor differences in details. For example, the mode change can be triggered by the external control task in the proposed method. An MTM-SDF task graph consists of a CIC task graph that follows the SDF semantics for each mode and an MTM. Figure 3-12 shows an example of a mode transition machine specification and it can be defined as follow:

Modes	Variables	Transitions		
Mode 1	Var 1	SrcMode	Conditions	DstMode
Mode 2	Var 2	Mode 1	Var 1 != 1 && Var 2 < 3	Mode 2
Mode 3	Var 3	Mode 2	Var 3 == 2	Mode 3

Figure 3-12. Mode transition machine specification

Definition 4.9 (Mode transition machine). Mode transition machine MTM consists of *Modes*, *Variables*, and *Transitions*.

- $Modes = \{m_k | k = 1, \dots, \# \text{ of modes} \}$ is a finite set of modes. (It corresponds to $Mode(t)$ in Definition 4.6.)
- $Variables = \{v_k | k = 1, \dots, \# \text{ of variables} \}$ is a finite set of variables.
- $Transitions = \{(m_{s_k}, C_k, m_{d_k}) | k = 1, \dots, \# \text{ of conditions} \}$ is a finite set of transition conditions.
 - m_{s_k} and m_{d_k} denote a source mode and a destination mode, respectively, and a condition C denotes a set of conditions.
 - Each condition $c \in C$ consists of $(v, comp, val)$ where $v \in Variables$ is a variable, $comp$ denotes a comparator ($=, \neq, \geq, \leq, >, <$) and val denotes a value.

3.5.2 Task graph specification

As mentioned in Chapter 2, a CIC task graph can be composed hierarchically, and a second-level graph is specified with an SDF graph. Therefore, a second-level graph can be specified with an MTM-SDF graph, so Definition 3.3 should be extended to consider an MTM-SDF graph.

Definition 4.10 (Port rate extension of the second-level task graph). In the CIC model, a second-level task graph is described by a tuple $(T, C, D) \times MTM$

- For each port $p \in Port(t)$, it has own rate $Rate(p, m) \in \{0\} \cup \mathbb{N}$ in each mode $m \in Modes$.

The internal behavior of a task should be defined manually depending on the mode of the system if it has a different behavior in each mode. Figure 3-13 shows an example code skeleton for the internal definition of a CIC task in the MTM-SDF

```

TASK_GO{
    Mode = SYS_REQ(GET_CURRENT_MODE_NAME);
    if Mode == "S1":
        receive data from input channel;
        code for functional behavior in mode S1;
        send data to output channel;
    else if Mode == "S2": ...
    if specific conditions:
        SYS_REQ(SET_MTM_PARAM_INT, task_name, var_name, value);
}

```

Figure 3-13. CIC task code skeleton in an MTM-SDF graph

graph. A CIC task first checks the current mode of its MTM before the beginning of execution and performs a proper action for the current mode. If it meets a specific condition, it may change a mode variable in its MTM to change the mode of the parent graph. In Table 3-3, it shows a control API list for the MTM-SDF model.

Table 3-3. Control APIs for an MTM-SDF graph

APIs	Description
<i>SYS_REQ(GET_CURRENT_MODE_NAME);</i>	Get a current mode of the task
<i>SYS_REQ(SET_MTM_PARAM_INT, task_name, name, value);</i>	Set a variable as a specified value

3.5.3 Execution semantic of an MTM-SDF graph

Note that the change of the mode variable does not trigger the MTM mode transition immediately. The MTM is triggered at the beginning of the iteration of the SDF schedule. An SDF graph has a well-defined notion of an iteration; since each node knows how many times it should be executed within an iteration, it can identify the iteration boundary by maintaining its execution counts autonomously.

We automatically generate a prelude code that manages the execution counts for each task to this end. The mode change is signaled to the CIC tasks with the iteration count. There is an exception for the source task of the MTM-SDF graph. A source task is a task that has no input channel inside the MTM-SDF graph. In case of a stream-based application, it commonly occurs that the first task receives the header frame that contains the information how to process the payload data. In that case, after reading the header frame we have to determine the mode of operations. In the *H.264 decoder* application, the type of an input frame is determined after the parsing task completes its execution. In that case, the mode change occurs at the end of the source task execution. To account for such an exceptional case, we designate a special source task in the extended CIC model. Then, the actual mode change is made after the designated source task finishes its execution.

Figure 3-14 shows an execution timeline of an MTM-SDF graph. Because each iteration of the task graph can be overlapped for a pipelined execution, a mode transition history and execution count for each task should be managed for consistent execution of an MTM-SDF graph. Code generation for this will be explained in Chapter 5.2.

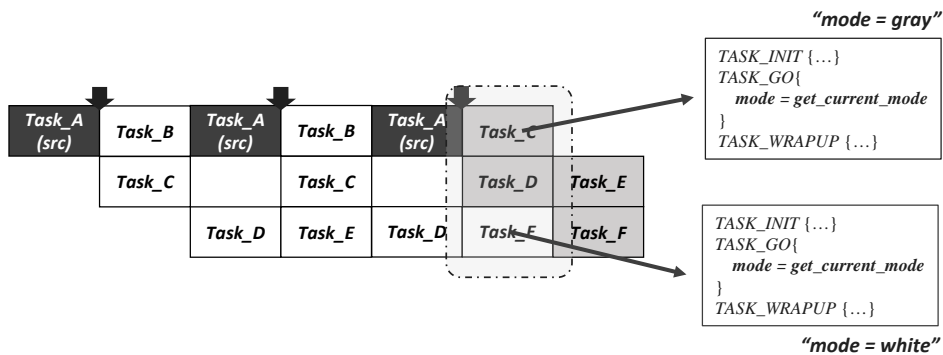


Figure 3-14. Timeline of the MTM-SDF graph execution

Figure 3-15 shows the MTM-SDF graph associated with the *H.264 decoder* task of Figure 3-3. The *H.264 decoder* task contains two modes: *I-frame* and *P-frame*. In *I-frame* mode, it does not perform inter prediction. So, the sample rates on all channels connected with *InterPredY/U/V* tasks are all set to 0. But, to perform inter prediction in the next *P-frame* mode, it needs to store previous frames in *InterPredY/U/V* tasks. So, sample rates for input channels from *Decode* tasks to *InterPredY/U/V* tasks will not be changed. In *P-frame* mode, since it does not perform intra prediction, the sample rates for all connected input/output channels with *IntraPredY/U/V* tasks are set to 0. In *H.264 decoder* algorithm, the mode of current iteration should be determined after reading the frame information in the *ReadFileH* task. So, we designate it as a special source task.

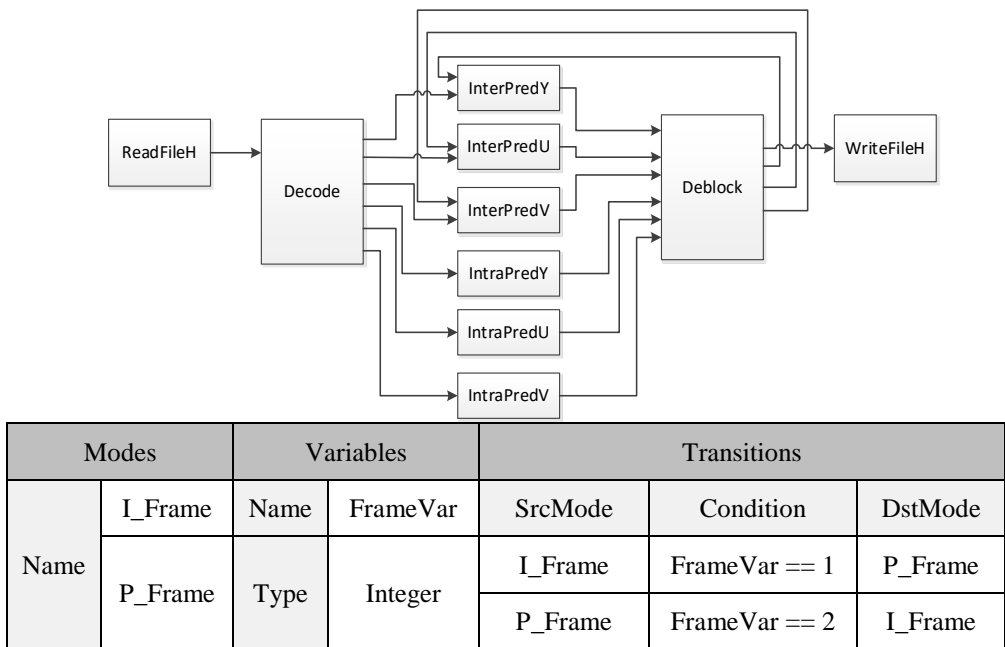


Figure 3-15. An MTM-SDF graph specification of H.264 decoder application

The MTM-SDF is basically equivalent to FSM-SADF. But, to our best knowledge, our specification is unique in that we use different specification methods for inter- and intra-application dynamism. Through such two-level specification, we are able to not only specify various kinds of dynamic behavior, but also to perform compile time analysis and system level optimization.

Chapter 4 Multiprocessor Scheduling of an Multi-mode Dataflow Graph

In system-level, the proposed task model supports an FSM-based specification, so we can extract a set of MTM-SDF graphs and the given timing requirements for each use case of the specified system. Also, in the proposed task model, each application is specified with an MTM-SDF model which supports dataflow model based analysis. Because the proposed task model supports formal model-based specification in both system-level and application-level, we can perform the design space exploration for the resource requirement of the system. For each use case, we assume a gang scheduling for each MTM-SDF graph, so we can calculate the number of required processors for each use-case as the sum of the number of required processors for each MTM-SDF graph. Then, the resource requirement of the specified system can be defined as the maximum number of required processors among all use-cases. Therefore, we need to analyze and minimize the resource requirement of an MTM-SDF graph for the given timing requirement.

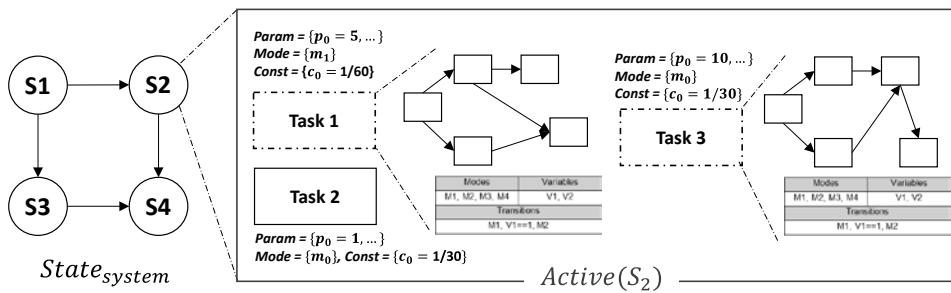


Figure 4-1. Extracted behavior of the entire system from the specification

4.1 Related work

There exist several extensions of the synchronous dataflow model that have been proposed to express the dynamic behavior of an application. Table 4-1 summarizes well-known dataflow-based extensions supporting dynamic behavior specification, compared with the proposed technique.

One of the most representative multi-mode dataflow models is FSM-SADF. As previously mentioned, in the FSM-SADF model, an application consists of multiple scenarios (modes) and each scenario is specified by an SDF graph. To specify multiple scenarios and their transitions, it defines a special control task called detector that has an FSM inside. The detector task sends the control information to the normal computation tasks that may change its behavior. For the FSM-SADF

Table 4-1. Comparison of various MoCs supporting dynamic behavior specification

MoC	Base model	Multiprocessor scheduling	Mode transition delay	Task migration
FSM-SADF	FSM, SADF	Static mapping/ scheduling	System reconfiguration overhead	Not allowed
PSDF	SDF	Not supported	Not considered	Not allowed
MCDF	SDF	Static mapping/ scheduling	Time interval between modes	Not allowed
VPDF	VRDF, CSDF	Dynamic scheduling	Not considered	Not allowed
MADF	SADF, VPDF	Real-time scheduling (fixed mapping)	Time interval between modes	Not allowed
BPDF	SDF	Not supported	Not considered	Not allowed
HDF	FSM, DF	Not supported	Not considered	Not allowed
Proposed	FSM, SDF	Static mapping/ scheduling	Time interval considering task migration	Allowed

model, several techniques to statically analyze the timing behavior such as worst case latency and throughput [65][66] have been proposed. Also, a binding-aware scenario graph [67][68] has been proposed to take into account the resource constraint. And, in [69], it considers reconfiguration overhead for DVFS (Dynamic Voltage Frequency Scaling) as the mode transition delay. However, it only considers the worst-case performance analysis of the FSM-SADF graph for the given task mapping, and requires inherently exponential time-complexity for exact analysis. The authors of [67] proposed a predictable design flow which finds Pareto-solutions among resource requirements under a given throughput constraint. But it does not allow task migration between different modes, and it constructs a static schedule for each mode independently.

PSDF (Parameterized Synchronous Data Flow) [8][9] proposes a meta-modeling technique for run-time adaptation of parameters in a structured way. In the PSDF model, the dynamic behavior of a task is modeled by parameters and the task behavior can change at the iteration boundary at run-time. Since the PSDF becomes an SDF graph at each iteration, it can be regarded as a multi-mode dataflow graph that may change modes every iteration. To the best of our knowledge, however, there is no research published for multiprocessor scheduling of the PSDF model.

MCDF (Mode-Controlled Data Flow) [70] is another extended model that can express the data-dependent functional behavior. The authors of [71] proposes temporal analysis techniques for a given sequence of mode changes. In addition, they propose a quasi-static multiprocessor scheduling technique under the given constraints. Based on static scheduling information of each mode, they model the mode transition interval which denotes a time interval between two consecutive schedules. While the modeling method is similar to our proposed technique, it does

not allow task migration between modes. Thus the task migration delay in their work does not take into account the mode transition delay.

VRDF (Variable-Rate Data Flow) [72] is proposed to allow variable port rates within a specified range, and VPDF (Variable-rate Phase Data Flow) [73] is proposed to combine characteristics of VRDF and CSDF where each actor has a sequence of phases, and for every phase, the number of firings can be parameterized. For these MoCs, buffer size analysis technique is proposed to satisfy the given timing and resource constraints. But it assumes dynamic scheduling, and to the best of our knowledge, no technique has been proposed for static scheduling of the VPDF graph.

MADF (Mode-Aware Data Flow) [74] has been proposed to support hard real-time scheduling for multi-mode CSDF (Cyclo-Static Data Flow) model [75]. It combines advantages of SADF and VPDF to specify application level dynamism. Also, it proposes MOO (Maximum-Overlap Offset) mode transition protocol allowing overlapped execution between modes, similarly to the proposed technique. With this mode transition protocol, the temporal behavior of individual modes and during mode transitions can be analyzed independently. But it assumes that the mapping of tasks is given and does not allow task migration between modes.

BPDF (Boolean Parametric Data Flow) [78][79] supports change of port rates and graph topology at run-time using integer and boolean parameters. In BPDF model, integer parameters are used to change port rates at each iteration, and boolean parameters are used for activation and deactivation of edges to change graph topology. It constructs a parallel ASAP scheduling on a many-core platform in compile-time. But it assumes that each task is mapped onto a separate processing element of the many-core.

HDF (Heterogeneous Data Flow) (or *-chart) [50] supports multimode applications through an FSM that executes an iteration of an SDF graph in each state. So, an application is specified with a set of different SDF graphs combined with an FSM. To the best of our knowledge, there is no research result published for multiprocessor scheduling of the HDF model.

While various analysis and scheduling techniques have been proposed for those MoCs, no one considers task migration between modes in the application-level. In [83], task migration is considered in the failure-aware task scheduling technique where an SDF graph is scheduled multiple times with different number of processors allocated, aiming to maximize the throughput with the allocated number of processors. When a processor fails in the middle of execution, it changes the schedule that uses the reduced number of processors by one. Then task migration occurs between two different schedules before and after processor failure. They try to minimize the migration cost between two SDF schedules. This method is similar to the base method that will be used for comparison in this dissertation: schedule each mode separately and find the best processor-to-processor mapping (or processor renaming) in order to minimize the migration cost. In summary, to the best of our knowledge, this dissertation is the first work which proposes a multiprocessor scheduling technique of an MMDF graph allowing task migration between modes, and analyzes the throughput requirement considering the mode transition delay.

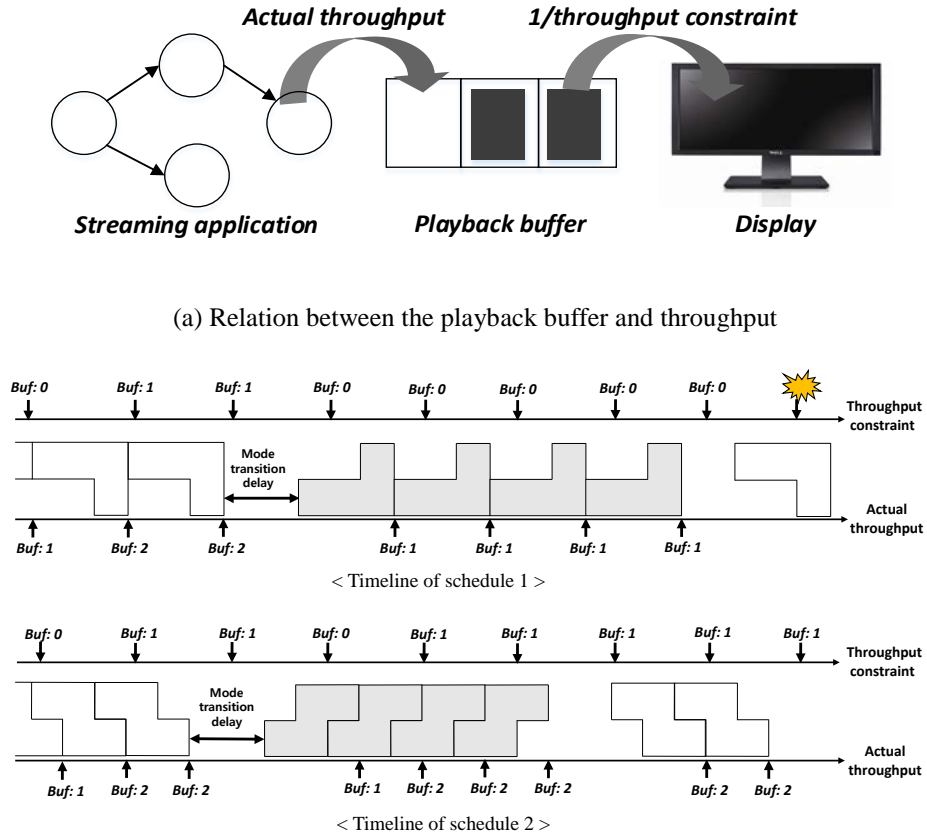
Several researches have been performed to explore the design space for multi-mode applications. But the meaning of “mode” is different from what is assumed in this dissertation. In [81] that introduced a hardware-software cosynthesis technique for multi-mode multi-task embedded systems, a mode is

defined as a set of SDF graphs. They schedule several SDF graphs together by considering the resource sharing. Their work is similar to [82] that designs MPSoCs to meet the throughput constraints of a set of applications while minimizing the resource requirements. With the different definition of mode, they did not consider the task migration overhead in their scheduling technique. Task migration is considered in [59] that proposes a hybrid design-time/run-time strategy for mapping software specified by a set of KPN graphs onto heterogeneous architectures. Even though they define a mode as a set of KPN graphs, they consider migration costs when mapping the processes onto the processors. To minimize the migration cost, they try to keep the mapping decision of an application over all scenarios (modes).

In summary, to the best of our knowledge, it is the first work which proposes a multiprocessor scheduling technique of a multi-mode dataflow graph and analyzes the throughput requirement and the buffer size, considering the mode transition delay.

4.2 Motivational example

4.2.1 Throughput requirement calculation considering mode transition delay



(b) Throughput requirement considering the mode transition delay

Figure 4-2. Motivational example of throughput requirement calculation considering the mode transition delay

In a streaming application that requires the periodic output stream production, an output buffer is usually used as depicted in Figure 4-2 (a). The display update

frequency becomes the throughput constraint of the application. If the execution time of an application varies dynamically, the time interval between output samples will vary even though the average throughput satisfies the throughput constraint. The output buffer is used to tolerate such variation to produce the periodic output stream to the display. The size of output buffer depends on the amount of variation.

Suppose that a streaming application has multiple modes of operation and each mode is specified by an SDF graph. For each mode of operation, we can find a static schedule of the associated SDF graph that satisfies the throughput requirement. If the static scheduling is followed at run-time, the output samples will be produced periodically without variation of output intervals at each mode. When mode change occurs, however, the interval between two output samples may vary due to additional time delay during mode transition. Then the overall throughput may become smaller than the throughput constraint even if the throughput of each mode is no less than the throughput constraint, as illustrated in Figure 4-2 (b).

Figure 4-2 (b) shows two different schedules of an MMDF graph that consists of two different modes of operation. The arrows on an upper line represent the times when the system dequeues data from the output buffer periodically with the same rate as the throughput constraint. The arrows on a lower line tell when an MMDF application enqueues data to the output buffer. A number annotated on an arrow, $Buf:x$, denotes the number of data items in the output buffer after the access is completed. If the number becomes negative, it means that buffer underflow occurs.

In case of schedule 1 in Figure 4-2 (b), even though the schedule of each mode satisfies the throughput constraints, the throughput constraint is eventually violated since the mode transition delay is accumulated. To avoid this problem, we need to set the throughput constraint of each mode tighter than that of the application as

schedule 2 illustrates in the figure; it keeps the throughput constraint because it fills the output buffer faster than the throughput constraint. Therefore we need to calculate the actual throughput requirement for each mode considering the mode transition delay, in order not to violate the given constraint.

4.2.2 Task migration between mode transition

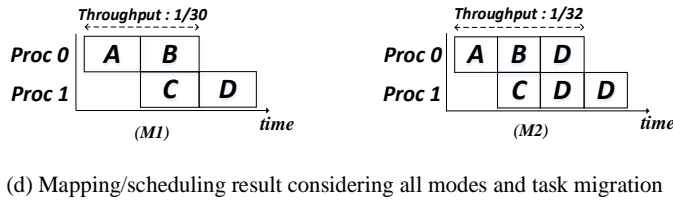
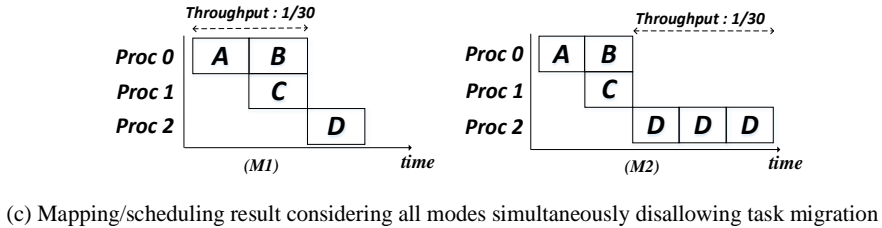
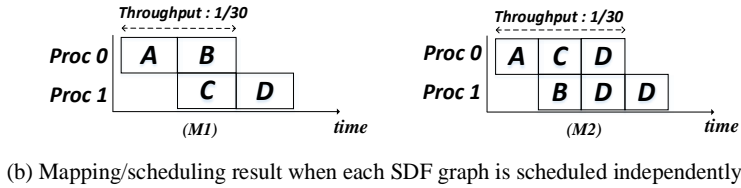
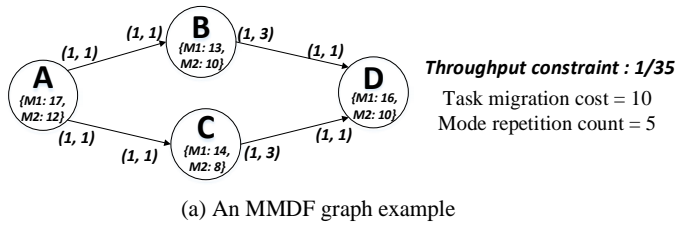


Figure 4-3. Motivational example of task migration

Figure 4-3 (a) shows an MMDF graph example which consists of two modes: M1 and M2. We assume that the throughput constraint of the MMDF graph is given as $1/35$. Each execution mode of an MMDF graph is represented with an SDF graph. The execution time and sample rates of each node may vary depending on the execution mode. In mode M1, the execution times of nodes A, B, C, and D are 17, 13, 14, and 16, respectively and in mode M2, they are 12, 10, 8, and 10. The output sample rates of nodes B and C are unity in mode M1 while they are 3 in mode M2. Refer to the next chapter for the formal description of the MMDF model assumed in this dissertation. Also, in this chapter, we only consider the task migration overhead as the mode transition delay to simply show the effect of task migration during mode transition.

A naive approach to schedule an MMDF graph is to schedule an SDF graph in each mode independently with multiple objectives of resource minimization and throughput maximization. For example, for the given throughput constraint, we find an optimal mapping/scheduling result in each execution mode as shown in Figure 4-3 (b). Since it does not consider mapping results in the other modes, a node may be mapped onto different processors between modes. Therefore, the mapping result requires task migration when the mode changes. In Figure 4-3 (b), nodes B, C and D will be migrated to other processors when the mode transition occurs.

Another approach to schedule an MMDF graph is to consider all modes simultaneously disallowing task migration [67][70]. Since the mapping is constrained in these approaches, the scheduling results generally require more processors than those that allow task migration. For instance, three processors are required to meet the given throughput constraint for the mapping/scheduling result without task migration as shown in Figure 4-3 (c), while two processors are enough

for the scheduling result with task migration in Figure 4-3 (b). Since the objective of the proposed scheduling framework is to minimize the resource requirement under a given throughput constraint, the proposed approach allows task migration. Their approach is used as a reference technique for comparison with the proposed technique in experiments.

Consider the former approach that allows task migration in Figure 4-3 (b). If the mode transition occurs frequently and the task migration overhead is non-negligible, then the given throughput constraint may not be satisfied. For instance, assume that the mode transition occurs every 5 iterations and the task migration overhead of each node is 10. In Figure 4-3 (b), 30 time unit is added every 5 iterations because nodes B, C and D should be migrated for mode transition. Then, the output buffer will be eventually empty, because the average throughput performance of the MMDF graph becomes lower than the throughput constraint.

Therefore, in this dissertation, we propose another approach that schedules the SDF graphs of all modes simultaneously allowing task migration among execution modes. Figure 4-3 (d) shows a mapping and scheduling result produced by the proposed technique. It requires 2 processors and only 10 additional time units for task migration, which may satisfy the throughput requirement with proper output buffering. Throughput analysis considering task migration overhead will be discussed in the Chapter 5.3.

4.3 Problem definition

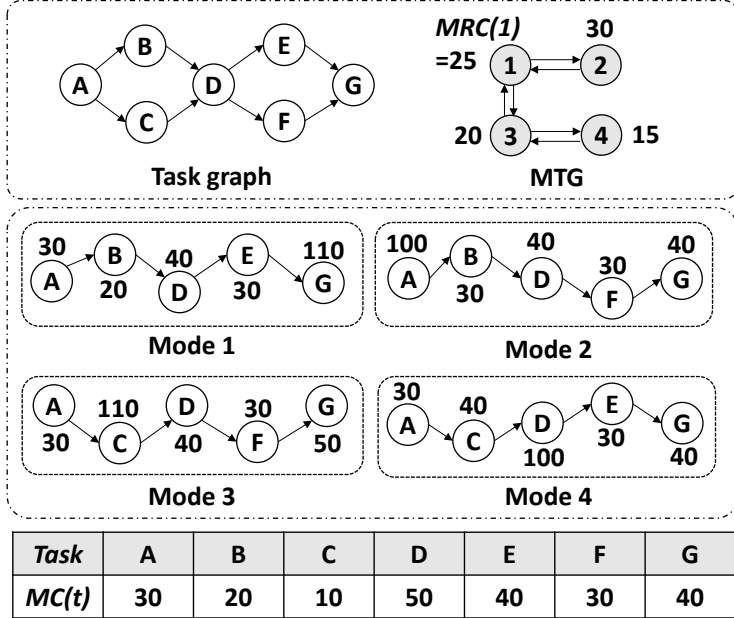


Figure 4-4. An MMDF graph example

The MMDF model assumed in this dissertation is not a specific model but a generic model encompassing existing similar models such as FSM-SADF [10] and MTM-SDF. In those models, the mode transition is specified by an FSM and all modes are integrated into a single SDF graph with varying configuration parameters. Figure 4-4 shows an MMDF graph example. We first define the MMDF model and the problem formally.

Application model: An MMDF graph is specified by a combination of a task graph and a mode transition graph (MTG), or $(T, C, D) \times MTG$, where

- MTG is specified by a tuple $(Mode, Trans)$ where $Mode$ is a finite set of modes and $Trans$ is a finite set of transitions. $Trans$ is specified as follows: $Trans =$

$\{(m_p, m_n) | m_p \in Mode, m_n \in Mode\}$ where m_p denotes a previous mode and m_n denotes a next mode.

- T is a finite set of computational tasks, and T_m denotes a subset of T which contains tasks executed in mode m . Each task $t \in T$ has a set of ports P_t to send/receive data to/from other adjacent tasks. $P_t = IP_t \cup OP_t$ where IP_t is a set of input ports and OP_t is a set of output ports. For each port $\in P_t$, it is assigned a fixed rate, $Rate(\rho, mode)$, in each execution mode. If a port rate is one for all modes, it is omitted for simple illustration of figures. Then the graph becomes an SDF graph for each mode.

- C is a finite set of FIFO channels. A channel defines a one-to-one connection between two end ports. For each channel $c \in C$, $srcTask(c)$ and $dstTask(c)$ denote a source task and a destination task of channel c , respectively.

- D is a set of the number of initial tokens in all channels. $d_c(m) \in \{0\} \cup \mathbb{N}$ for $\forall d_c \in D$ is the number of initially stored tokens in the channel c in mode m .

Architecture model: the target architecture consists of homogeneous processing elements.

- PE is a set of processing elements. For each $p \in PE$ and $m \in Mode$, $Map(m, p) = \{t | t \in T \text{ where } t \text{ is mapped onto a processor } p \text{ in mode } m\}$

Note that even though the proposed technique is applicable to heterogeneous multiprocessor systems, this dissertation assumes a homogeneous multiprocessor system for simple explanation and implementation.

To analyze the scheduling performance of an MMDF graph, we assume profiling information is available as follows:

Profiling information:

- Worst case execution time (*WCET*) for each task $t \in T$ and $m \in Mode$ is given as $WCET(t, m, p)$ for each processing element $p \in PE$ of the target architecture. In Figure 4-4, the *WCET* of a node is annotated in each mode. For example, *WCET* of node A is 30 in modes 1, 3, and 4, and 100 in mode 2.
- For each $m \in Mode$, we are given a minimum number of iterations that the application stays at the mode, which is denoted by $MRC(m)$ where *MRC* stands for the minimum repetition count. As *MRC* becomes smaller, the mode transition occurs more frequently. A mode is associated with an *MRC* value as shown in Figure 4-4 where *MRC* is 25, 30, 20 and 15 in modes 1, 2, 3, and 4, respectively.
- For each $t \in T$, task migration cost is given by $MC(t)$. If the system is a distributed memory system, the migration cost will include the time overhead of moving the code and the context of a task between two processors. If it is a shared memory system, the migration cost will be small as cache miss penalty for the first reference (called *cold miss penalty*) of the task. A table in Figure 4-4 shows the *MC* value of each task. For instance, *MC* of node A is 30.

Execution semantic:

- (*Intra-mode operation*) In each mode, the associated SDF graph starts an iteration from the beginning. For a consistent SDF graph, there will be no change in the channel buffer state before and after an iteration.
- (*Mode switching decision*) To make a mode switching decision, a special type of task is usually designated in existing approaches (e.g. detector actor in [10], control actor in [70]). Similarly, we designate a specific task, called *mode decision task* (denoted by $t_{mode} \in T$) that determines the current mode of operation in the

MMDF graph. It implies that the set of predecessor tasks of the mode decision task is common to all modes. There is at most one mode decision task in the MMDF graph. Note that the mode switching decision can be made any time during an iteration depending on the scheduling order of tasks. If t_{mode} is not specified, the mode changing decision is assumed to be made before the start of an iteration.

- (*Mode transition*) Because we allow overlapped execution of two modes, we assume that mode transition is made at the iteration boundary: When a mode change is requested, the previous iteration of the SDF graph continues to finish its execution. In case the mode decision task is executed more than once in an iteration, we assume that the mode switch decision is made at the last instance of the mode decision task.

- (*Inter-mode consistency*) The channel buffer state is preserved when a mode transition occurs. Initial tokens of a channel in the current mode are transferred to the initial tokens of a channel in the next mode. It incurs additional scheduling dependency between task executions in two modes: In case there exist initial tokens in channels, the start of the next mode should be delayed until the same number of tokens are produced in the current mode.

We assume that similar models introduced in Chapter 4.1 can be transformed to the MMDF model if it can keep the aforementioned construction rules and execution semantics. With those application and architecture models, profiling information and execution semantic, the problem addressed in this dissertation is summarized as follows:

PROBLEM: Find a mapping and scheduling result of an MMDF graph which satisfies the given throughput constraint

minimize. the number of required processors

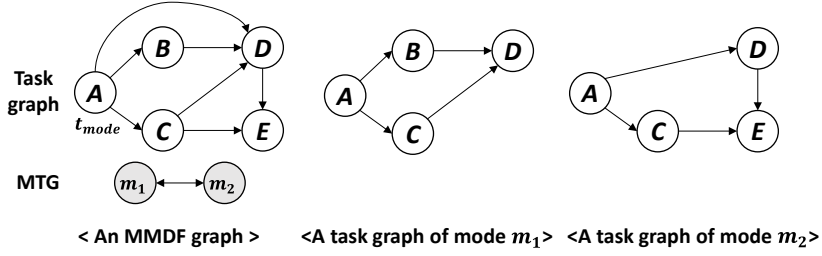
subject to. the overall throughput performance of the MMDF graph should be higher than the given throughput constraint.

The proposed MMDF scheduling framework is based on a genetic algorithm [98]. So, it needs to evaluate all candidate solutions in every iteration. How to evaluate whether a given mapping and scheduling result of an MMDF graph satisfies the given throughput constraint will be explained in the next chapter.

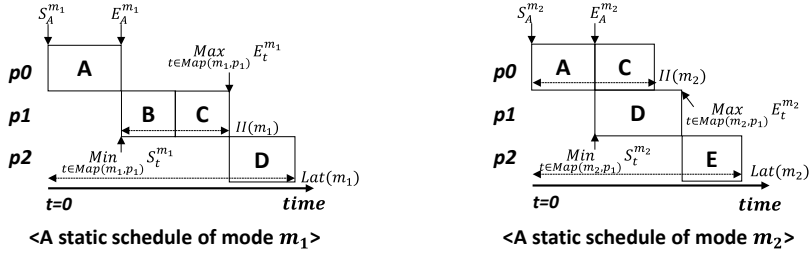
4.4 Throughput requirement analysis

Figure 4-5 (a) shows a simple MMDF graph example that consists of two modes of operation. For each mode, a static schedule which satisfies the given throughput constraint is constructed as shown in Figure 4-5 (b). If a mode transition does not occur, the schedule of the current mode will be repeated and the output samples will be produced periodically. The period is equal to the inverse of the throughput performance, which is denoted as the initiation interval (II) in the figure. If a mode transition occurs, production of the next output sample will be delayed because of interference between two different schedules of the previous mode and the next mode. Figure 4-5 (c) shows a timeline of the execution of the MMDF graph.

Even though the static schedule of each mode satisfies the given throughput constraint, the overall throughput performance of the MMDF graph may not satisfy the throughput constraint because of the mode transition delay. To satisfy the throughput constraint, we may need to tighten the throughput requirement for each mode of operation. To understand how the mode transition delay to the throughput requirement, we first explain how to compute the mode transition delay.



(a) An MMDF graph example



(b) Static schedule results of mode m_1 and m_2

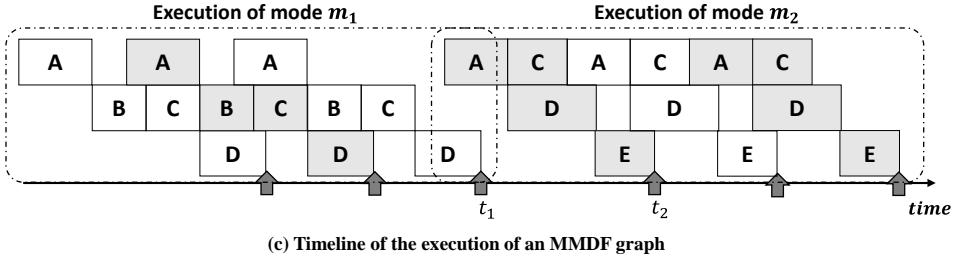


Figure 4-5. An MMDF graph example and static scheduling result

4.4.1 Mode transition delay

The mode transition delay between two modes is defined how the time interval between the last output production time of the previous mode (t_1 in Figure 4-5 (c)) and the first output production time of the next mode (t_2 in Figure 4-5 (c)) is larger than the initiation interval of the next mode. Suppose that the last iteration of the previous mode is started at $t = 0$. First we formulate the start offset (χ) of the first iteration of the next mode. The start offset (χ) is determined by the following three factors:

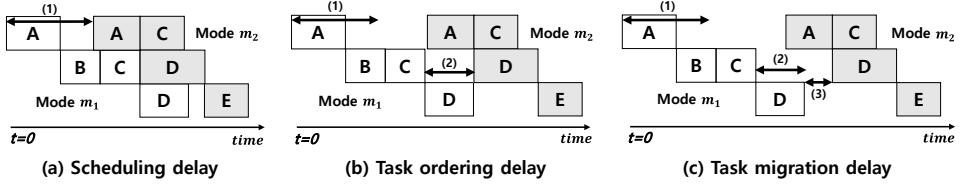


Figure 4-6. Mode transition delay between static schedules of mode m_1 and m_2 in Figure 4-5 (b)

1) *Scheduling delay* (D_{sched}): To guarantee consistent execution of the MMDF graph, we need to shift the start time of the subsequent mode. The time interval, denoted by (1) in Figure 4-6 (a), illustrates the scheduling delay between modes m_1 and m_2 of Figure 4-5.

There are three factors that determine the start time of the next mode. The first factor is the time delay (D_{proc}) to keep the temporal property of the static schedule of the next mode. The second factor (D_{mode}) is the finish time of the last instance of the mode decision task in the previous mode since the next mode may start afterwards. The third factor (D_{data}) accounts for the data dependency between two modes. In case there exist initial tokens in channels, the start of the next mode should be delayed until the same number of tokens are produced in the previous mode. The scheduling delay can be calculated as follows:

Definition 5.1 (Scheduling delay from mode m_i to mode m_j).

$$D_{sched}^{m_i m_j} = \max(D_{proc}^{m_i m_j}, D_{mode}^{m_i m_j}, D_{data}^{m_i m_j})$$

$$D_{proc}^{m_i m_j} = \max_{\substack{p \in PE \\ Map(m_i, p) \neq \emptyset \\ \wedge Map(m_j, p) \neq \emptyset}} \begin{cases} \max_{t \in Map(m_i, p)} E_t^{m_i} - \min_{t \in Map(m_j, p)} S_t^{m_j} & \text{if } \max_{t \in Map(m_i, p)} E_t^{m_i} - \min_{t \in Map(m_j, p)} S_t^{m_j} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$D_{mode}^{m_i m_j} = E_{t_{mode}}^{m_i}$$

$$D_{data}^{m_i m_j} = \max_{\substack{c \in C \\ d_c(m_j) > 0}} \begin{cases} E_{srcTask(c)}^{m_i} - S_{dstTask(c)}^{m_j} & \text{if } E_{srcTask(c)}^{m_i} - S_{dstTask(c)}^{m_j} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where E_t^n denotes the end time of task t in mode n , and S_t^m denotes the start time of task t in mode m . In Figure 4-5 (b), the start time and the end time of a task are depicted.

2) *Task ordering delay* (D_{order}): Because the proposed technique allows task migration between modes, a task can be mapped onto different processors in each mode. So two consecutive executions of the same task should not be overlapped or inverted during mode change. Therefore, we need to guarantee that a task in the next mode can start only after the task finishes its execution in the current mode. In Figure 4-6 (a), two executions of task D are overlapped between modes. Thus the execution of the next mode should be delayed by the task ordering delay denoted by (2) in Figure 4-6 (b). In case of task C , even though it is mapped onto different processors in each mode, no overlapping occurs as shown in Figure 4-6 (a). So task C does not incur additional time delay. The task ordering delay (D_{order}) can be formulated as follows:

Definition 5.2 (Task ordering delay from mode m_i to mode m_j).

$$D_{order}^{m_i m_j} = \begin{cases} \max_{t \in T_{m_i} \cap T_{m_j}} E_t^{m_i} - (S_t^{m_j} + D_{sched}^{m_i m_j}) & \text{if } \max_{t \in T_{m_i} \cap T_{m_j}} E_t^{m_i} - (S_t^{m_j} + D_{sched}^{m_i m_j}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

3) *Task migration delay* (D_{mig}): Tasks which are mapped onto different processors between modes should be migrated during the time interval between the end time in the previous mode and the start time in the next mode. If the time interval is not long enough to migrate the task, additional time delay will be required. In Figure 4-6 (b), task D should be migrated to other processor after the end of execution in the previous mode. However there does not exist enough time interval for task migration. So, additional time delay is needed, which is the task migration delay denoted by (3) in Figure 4-6 (c). In case of task C , additional time delay is not required because there already exists enough time for task migration. The task migration delay (D_{mig}) can be formulated as follows:

Definition 5.3 (Task migration delay from mode m_i to mode m_j).

$$D_{mig}^{m_i m_j} = \max_{\substack{t \in T_{m_i} \cap T_{m_j}, \\ proc(t, m_i) \neq proc(t, m_j)}} MC(t) - (S_t^{m_j} + D_{sched}^{m_i m_j} + D_{order}^{m_i m_j} + E_t^{m_i})$$

$$if \ S_t^{m_j} + D_{sched}^{m_i m_j} + D_{order}^{m_i m_j} + E_t^{m_i} < MC(t)$$

where, $proc(t, m_i)$ denotes a processor that task t is mapped onto in mode m_i .

Summing up all three types of delay mentioned above, we compute the start offset of the next mode as follow:

Definition 5.4 (Start offset of mode m in the case of mode transition $m_i \rightarrow m_j$).

$$\chi^{m_i m_j} = D_{sched}^{m_i m_j} + D_{order}^{m_i m_j} + D_{mig}^{m_i m_j}$$

Remind that the mode transition delay between two modes is defined how the time interval between last output production time of the previous mode and first output production time of the next mode is larger than the initiation interval of the next mode. Since the output production time of each mode equals to the latency of the static schedule, the mode transition delay can be formulated as follows:

Definition 5.5 (Mode transition delay from mode m_i to mode m_j).

$$\forall (m_i, m_j) \in Trans, TransDelay(m_i, m_j) = Lat(m_j) + \chi^{m_i m_j} - Lat(m_i) - II(m_j)$$

where $Lat(m_j)$ represents the latency of mode m_j and $II(m_j)$ represents the initiation interval of mode m_j .

Note that, if $Lat(m_j) + \chi^{m_i m_j} - Lat(m_i) - II(m_j) \leq II(m_j)$, then $TransDelay(m_i, m_j)$ will be smaller than zero. It means that the time interval of the output production times during a mode transition can be shorter than the output production time interval of the next mode ($II(m_j)$).

The mode transition delay will be used to determine the required output buffer size and throughput requirement of an MMDF graph for the given throughput constraint. Details will be explained in the following chapter.

4.4.2 Arrival curves of the output buffer

To determine the buffer size and throughput requirement for each mode, we compute the arrival curves [103] of the input and the output streams in the buffer. The arrival curve of a stream informs the number of arriving (or departing) samples (y-axis) within a time interval (x-axis) as shown in Figure 4-7. For conservative estimation, we use the maximum arrival curve for the output stream and the minimum arrival curve for the input stream.

In Figure 4-2 (a), task *Display* dequeues data from the output buffer periodically with satisfying the throughput constraint, which is depicted as the output curve (gray solid line) in Figure 4-7. The slope of the output curve is equal to the throughput constraint.

The black solid line represents the minimum arrival curve of the input stream which presents the number of generated samples to the output buffer. The buffer size is computed based on the minimum repetition count (*MRC*), the inverse of the

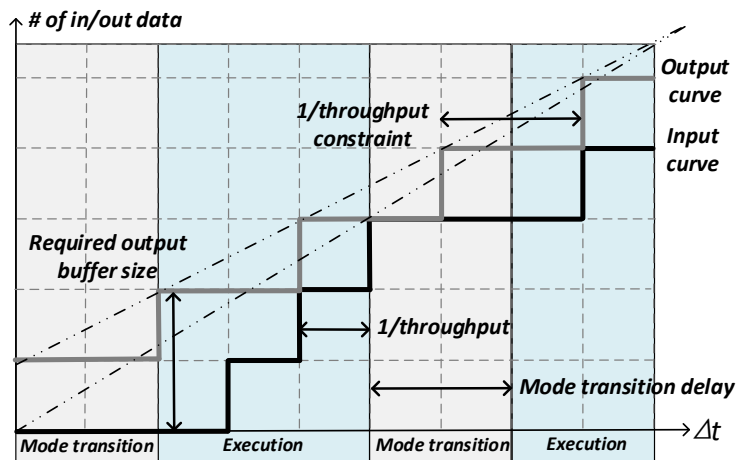


Figure 4-7. Arrival curves for input and output streams in the output buffer

throughput, and the maximum mode transition delay among all possible transition scenarios to the mode. The maximum mode transition delay to mode m is computed as following:

Definition 5.7 (Worst-case mode transition delay to mode m).

$$MaxTransDelay(m) = \max_{\substack{\forall (m_i, m_j) \in Trans, \\ m_j = m}} TransDelay(m_i, m_j)$$

Note that since the slope of the curve depends on the mode transition delay, the mode repetition count, and the throughput performance of the MMDF schedule, the buffer size is determined after constructing an MMDF schedule meeting the throughput constraints in all modes.

4.4.3 Buffer size determination

As discussed in Chapter 4.2.1, an output buffer is adopted to produce data samples periodically. Since the mode transition delay causes the jitter of output production in an MMDF application, the output buffer should be large enough to provide the data samples during mode transitions. The required output buffer size depends on the maximum mode transition delay and the throughput difference between the input stream and the output stream in the buffer. In each mode, we compute the buffer size and then choose the maximum buffer size in all modes.

From the arrival curves, we obtain the minimum output buffer size which is the maximum difference between the curves in every time interval (Δt). If the overall throughput constraint is satisfied, the output buffer size is computed as following.

Theorem 1. (Output buffer size) The minimum size of the output buffer to satisfy the given throughput constraint ($ThrConst$) is decided by the following equation:

$$Output\ buffer\ size = \lceil MaxInterval_{overall} \times ThrConst \rceil$$

$$where\ MaxInterval_{overall} = \max_{\substack{v(m_i, m_j) \\ \in Trans}} TransDelay(m_i, m_j) + II(m_j)$$

Proof. The buffer size is determined by the maximum distance between the input and the output curves, which is illustrated in Figure 4-7. Because the proposed technique uses a higher throughput requirement than the throughput constraint when constructing a static schedule of each mode, the slope of a tangent line of the input arrival curve during execution should be larger than that of the output curve. Consider another tangent line that connects the starting point of mode transition and the ending position of the current mode execution, which is shown with a double-dotted line in Figure 4-7. The slope of this tangent line cannot be smaller than that of the output curve in order to keep the buffer size finite. If it is smaller, the gap between two tangent lines will increase unboundedly if we apply the same mode transition repeatedly. By the construction rule of the arrival curves, the distance between the output curve and the input curve decreases as the time window increases. Therefore, the maximum distance between two curves occurs is obtained just before the first jump of the input curve. Therefore,

$$\begin{aligned} Output\ buffer\ size &= \left\lceil TransDelay_{overall} \div \frac{1}{ThrConst} \right\rceil \\ &= \lceil TransDelay_{overall} \times ThrConst \rceil \end{aligned}$$

Q.E.D.

4.4.4 Throughput requirement analysis

The overall throughput performance of the MMDF graph depends on the mode transition delay as well as how frequently mode transition occurs. Since mode transition is triggered by an internal/external events at run-time, it may not be possible to know the mode transition scenario at compile-time. Even though the exact mode transition scenario is not known, we assume that the minimum number of iterations is given as a part of the input information. Based on this information, we draw the input arrival curve of Figure 4-7 and estimate the buffer size conservatively.

Now we compute the throughput requirement of each mode. For conservative estimation, the input curve should be steeper than the output curve in all modes in Figure 4-7. The throughput requirement in each mode can be formulated as follows:

Theorem 2. (Throughput requirement) *The throughput requirement in mode m denoted as $ThrRequire(m)$, is formulated as following:*

$$ThrRequire(m) = \frac{ThrConst \times MRC(m)}{MRC(m) - (MaxTransDelay(m) \times ThrConst)}$$

Proof.

$$The\ slope\ of\ input\ curve = \frac{MRC(m)}{MaxTransDelay(m) + \frac{1}{ThrRequire(m)} \times MRC(m)}$$

$$And\ the\ slope\ of\ output\ curve = \frac{1}{1/ThrConst} = ThrConst$$

Since the slope of input curve should not be smaller than that of the output curve,

$$\frac{MRC(m)}{MaxTransDelay(m) + 1/ThrRequire(m) \times MRC(m)} \geq ThrConst$$

$$and\ ThrRequire \geq \frac{MRC(m)}{MRC(m)/ThrConst - MaxTransDelay(m)}$$

Q.E.D.

If the throughput performance of each mode is higher than the throughput requirement calculated by Theorem 5.2, the MMDF graph will satisfy the throughput constraint. Note that the throughput requirement is a conservative bound. In case we know the exact scenario of mode transitions, the computed buffer size and the throughput requirement will be tight bounds.

4.5 Proposed MMDF scheduling framework

4.5.1 Optimization problem

The proposed technique is to find a static schedule of each mode cooperatively in a single optimization framework based on a genetic algorithm. We aim to minimize the resource requirement of an MMDF graph while satisfying the given throughput constraint. As discussed in the previous chapter, the throughput requirement of each mode should be computed considering the mode transition delay. On the other hand, depending on the throughput requirement of each mode, resource requirement may vary. As the throughput requirement becomes tighter, more processing elements are likely to be required to satisfy the throughput requirement. Therefore the mode transition delay is an important factor that affects the resource requirement of the MMDF graph. As stated in Definition 5.4 and Definition 5.5, the mode transition delay is influenced by three types of delay (D_{sched} , D_{order} , and D_{mig}) as well as the individual schedule of each mode. Therefore, to minimize the mode transition delay, we need to consider those delays when constructing the schedule of each mode cooperatively.

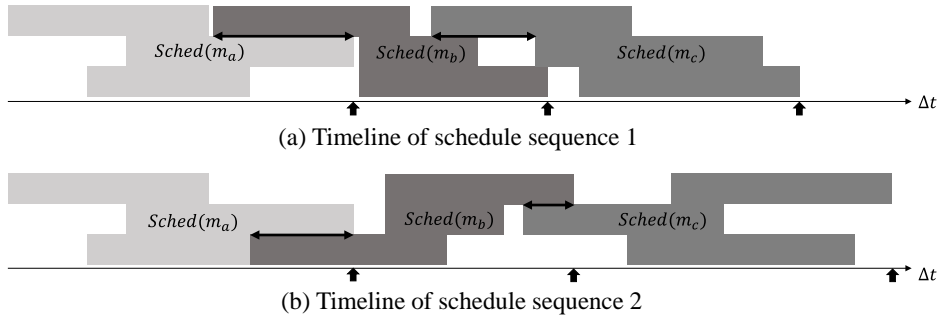


Figure 4-8. Overlapped schedule sequences among modes ($m_a \rightarrow m_b \rightarrow m_c$)

Figure 4-8 shows how the overlapped scheduling among different modes has an effect on the overall throughput performance of an MMDF graph. Even though the throughput performance of each mode is the same as shown in Figure 4-8 (a) and (b), the overall throughput performance of the case of Figure 4-8 (a) is better than the case of Figure 4-8 (b), because the schedules in Figure 4-8 (a) can be overlapped more than the case of Figure 4-8 (b). So the static schedule of each mode should be constructed considering the schedules of the other modes.

Multiprocessor scheduling of a dataflow graph is a well-known NP-hard problem. Our MMDF scheduling problem is much harder since an MMDF graph consists of a set of modes and each mode is specified by an SDF graph. Also we need to consider the mode transition delay and compute the throughput requirement of each mode dynamically. To tackle this problem, we adopt a meta-heuristic based on a genetic algorithm to find an approximate solution.

4.5.2 GA configuration

The overall GA procedure of the proposed framework is shown in Figure 4-9.

Initialization & Selection: Since a task (or node) can be mapped to different processors in modes, each task is regarded as a unit of mapping in each mode. The chromosome for GA is configured as shown in Figure 4-10. A chromosome is a set of mapping for each execution mode. Each gene of the chromosome represents to which processor a task in each execution mode is mapped. Chromosomes of initial population are randomly generated and selected for crossover and mutation. The number of selection is a configurable parameter of the GA framework.

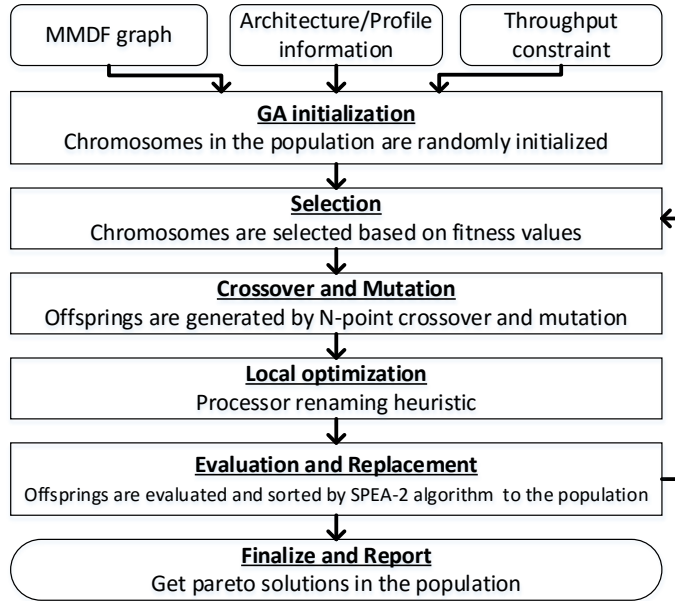


Figure 4-9. The overall GA framework

Crossover & Mutation: Crossover and mutation operations are applied to genes of each mode separately. As we explained in Chapter 4.3, the predecessor tasks of the mode decision task t_{mode} are common to all modes, so can be regarded as mode-independent. We do not change the mapping of those tasks among all modes.

Local optimization: To help the convergence of evolutionary process, a local optimization step is performed before the evaluation step. For local optimization, we devise a processor renaming heuristic that changes the processor id in each mode to reduce the migration cost. The details will be explained in the next chapter.

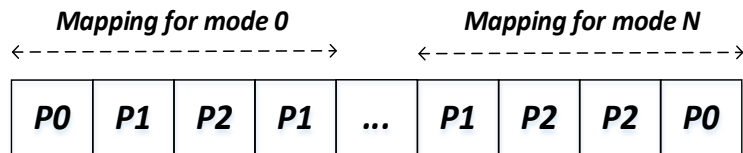


Figure 4-10. Chromosome structure

Evaluation & Replacement: In this step, we apply a list scheduling heuristic to find a static task schedule in each mode, based on the mapping information given by each chromosome. Once we construct a static schedule, we evaluate the fitness value of each offspring and check whether the throughput constraint is satisfied or not. Currently we do not consider inter-task interferences caused by shared resources such as cache and memory. However several researches [99][100] are already proposed to analyze the worst case contention scenario for a given task graph considering shared resource contentions. So if we need to consider inter-task interference, we can adopt those researches to the evaluation step of the proposed GA framework. The fitness function will be described in the next chapter. Chromosomes in the population are sorted by their fitness values and poor chromosomes are eliminated by SPEA-2 algorithm [101].

4.5.3 Fitness function

The objective of the MMDF scheduling is to minimize the number of processors. The required number of processors is defined as the maximum number of processors in all modes.

Definition 5.7. (The number of processors for an MMDF graph).

$$\text{The number of processors} = \max_{m \in \text{Mode}} |\text{Proc}_m|$$

$$\text{where } \text{Proc}_m = \{p \in PE \mid \text{Map}(m, p) \neq \emptyset\}$$

Since the large mode transition delay will degrade the throughput performance and more processors are likely to be required to meet the given throughput constraint, the mode transition delay including task migration overhead is considered to evaluate the number of required processors. And, the GA framework

also aims to minimize the overall task migration cost as the secondary objective. The reduction of task migration will save energy consumption of the system, and reduce the network traffic in an NOC architecture [102]. Therefore it is very desirable to reduce the total task migration cost of an MMDF graph considering all mode transition scenarios; the total task migration cost is defined as follows:

Definition 5.8. (Total task migration cost).

$$MigCost_{total} = \sum_{(m_p, m_n) \in Trans} MigCost(m_i, m_j)$$

$$where MigCost(m_i, m_j) = \sum_{p \in PE} \sum_{t \in \{Map(m_i, p) - Map(m_j, p)\}} MC(t)$$

We sum up the migration cost of all possible migration scenarios that are defined by the *MTG*. For each transition in the *MTG*, we accumulate the migration cost of all tasks that are mapped to different processors after the mode transition.

4.5.4 Local optimization technique

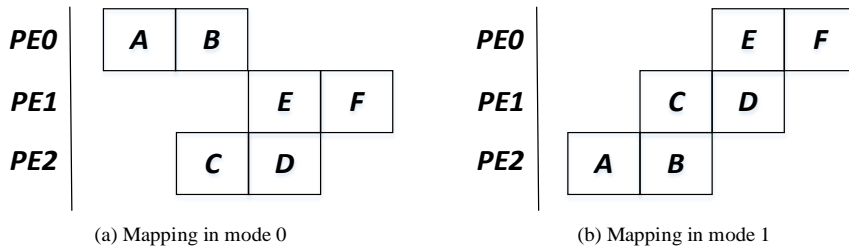


Figure 4-11. Without processor renaming, every task should be migrated when mode transition occurs. If PE0 in mode 0 is renamed to PE2 in mode 1, PE1 to PE0, and PE2 to PE1, no task migration is required

Figure 4-11 shows a motivational example for local optimization, where two modes have different task mappings defined in the chromosome and a mode transition from mode 0 to mode 1 occurs. In the mapping result, all tasks should be migrated. However, since the proposed technique assumes a homogeneous multiprocessor system, it is possible to rename the processor id in each mode, which is called *processor renaming*. If *PE0* in mode 0 is renamed to *PE2* in mode 1 then tasks *A* and *B* do not need to be migrated. Similarly, if *PE1* in mode 0 is renamed to *PE0* in mode 1, and *PE2* to *PE1* then no task migration is required. Without the processor renaming technique, good solutions such as Figure 4-11 will be evaluated as poor solutions due to high migration delay, which seriously hinders the convergence of GA.

for all mode transition scenarios:

```

    curr ← mapping information of src mode of the transition;
    next ← mapping information of dst mode of the transition;
    for all mapping information of each processor (cId) in curr:
        for all mapping information of each processor (nId) in next:
            similarity ← check similarity between curr[cId] & next[nId];
            if (similarity > maxSimilarity):
                maxSimilarity ← similarity;
                swapProcId ← nId;
    change mapping of next between cId & swapProcId;

```

Figure 4-12. Pseudo code of the processor renaming heuristic

The time complexity of the processor renaming algorithm is given as P^M where P denotes the number of processors and M is the number of mode transition scenarios. Therefore, we devise a simple greedy processor renaming heuristic as

shown in Figure 4-12 to reduce the time complexity. In the proposed heuristic, the time complexity becomes $O(P^2 \times M)$. Note that processor renaming is only applicable for homogeneous processor systems. The heuristic measures the similarity between processors. The similarity between processors is defined by how many tasks are mapped on both processors in common.

Definition 5.9. (Similarity between processors).

$$\text{For } (m_i, m_j) \in \text{Trans}, \text{Similarity}(p_k, p_l) = |\text{Map}(m_i, p_k) \cap \text{Map}(m_j, p_l)|$$

For each mode transition, processors in the next mode are renamed to the processors in the current mode with the maximum similarity. Even though the proposed heuristic does not consider all possible processor renaming scenarios and does not provide the optimal renaming result, it reduces the time complexity significantly while generating good quality solutions.

4.6 Experimental results

To prove the viability of the proposed framework, we experiment with five synthetic examples and five real applications: H.264 decoder, lane detection, vocoder [74], MP3 decoder [105], and printer pipeline [105]. All experiments have been performed on Intel Core i7-4790K 4.00GHz machine with 8GB main memory. Internal parameters of the GA framework are set as shown in Table 4-2. μ and λ denote the number of parents and offspring, respectively.

Table 4-2. Configuration of the GA framework

Population size	100
μ and λ	100
Probabilities of crossover/mutation	0.9
Maximum generations	30000

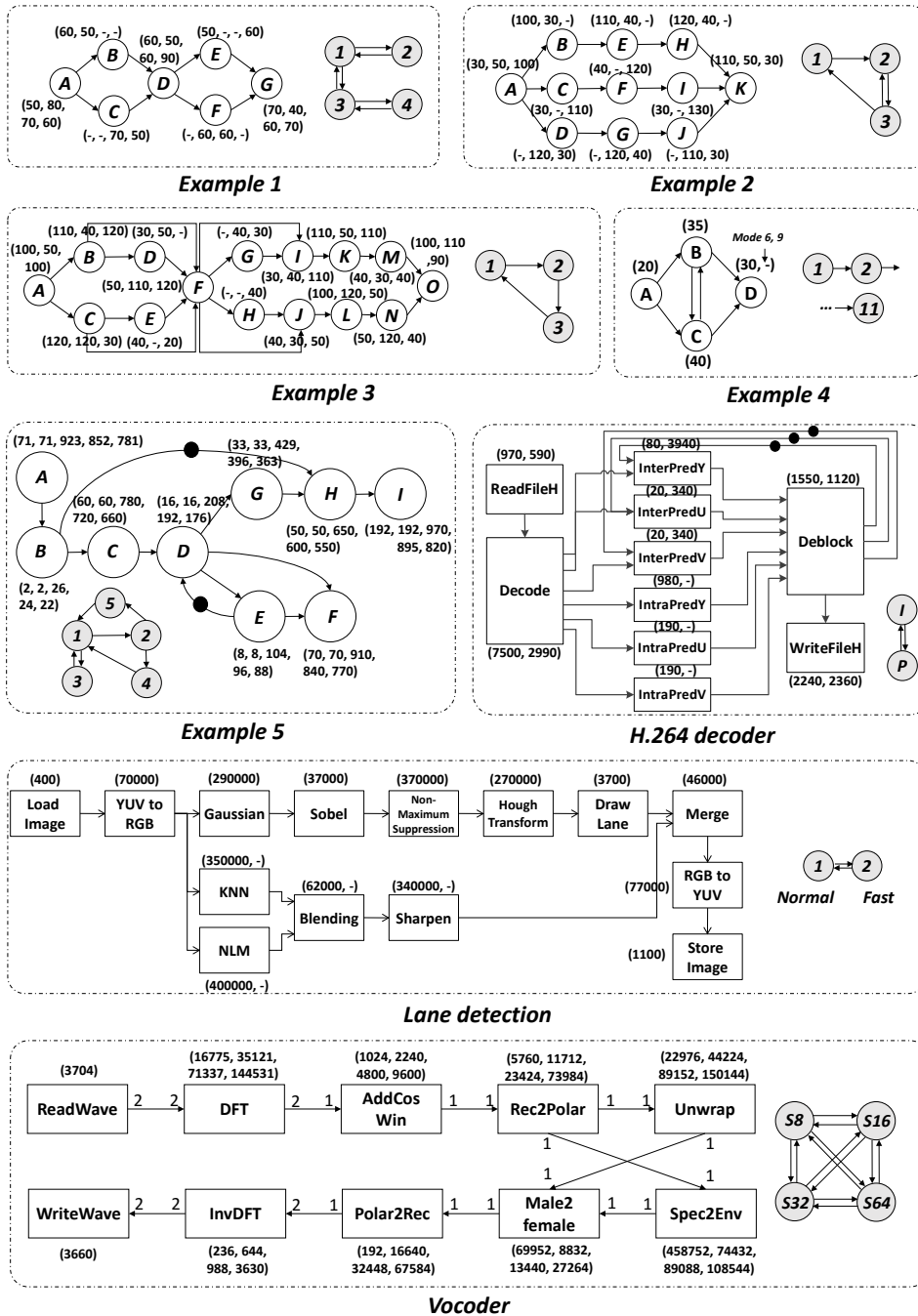


Figure 4-13. MMDF graph examples used in experiments

Figure 4-13 shows task graphs which are used for experiments. Task graphs of MP3 decoder and printer pipeline applications in [105] are omitted due to lack of space, but the task graph and the profiling information of each application are same with [105]. Also, $Rate(\rho, mode)$ for each port $\rho \in P_t$ is one if it is not specified. For the task graph of vocoder application in [74], we reduce the number of invocations for specific tasks (from *AddCosWin* to *Polar2Rec*) from 128 to 2 by clustering, so the given WCETs of those tasks in [74] are multiplied by 64. Also, we allow that each instance of the same node can be mapped onto different processors for data parallelism. The numbers above or under the tasks in Figure 4-13 indicate the $WCET(t, m, p)$ in each mode. In case that $WCET(t, m, p)$ of a task is constant in all modes, a single number is denoted. For synthetic examples, the WCET of each task is set to an arbitrary value, and the WCET of each task in the H.264 decoder and lane detection applications is set to profiled data in *us* unit. Also, for all examples, we assume that the source task of each task graph is the mode decision task t_{mode} which determines the mode of the current iteration.

4.6.1 MMDF scheduling technique

We compared the proposed technique with three different approaches listed in Table 4-3. The first approach, *Base*, schedules SDF graphs independently and performs the processor renaming heuristic, similarly with [83]. It is an iterative algorithm. For each mode, it constructs a set of Pareto-optimal solutions which are optimized with throughput and the number of processors, using a genetic algorithm. Then it selects an initial schedule that satisfies the throughput constraint with the minimum number of processors for each mode. Based on the mapping/scheduling results, it performs the processor renaming heuristic and adjusts the throughput requirement as discussed in the previous chapter, considering the mode transition

Table 4-3. Four approaches used in experiments

Approach	Consideration of other modes	Task migration	Mode-overlapped schedule
<i>Base</i>	Do not consider	Allow	Allow
<i>Fixed</i>	Consider	Do not allow	Allow
<i>Blocked</i>	Consider	Allow	Do not allow
<i>Proposed</i>	Consider	Allow	Allow

delay incurred by the initial schedules. If a schedule does not satisfy the calculated throughput requirement, it is replaced with another schedule which uses one more processor. Unless all scheduling results satisfy the newly adjusted throughput requirement in all modes, it repeats the mapping/scheduling with the new adjusted throughput requirement until the mapping/scheduling results satisfy the adjusted throughput requirement. Comparison with *Base* approach will show the reason why all modes should be scheduled simultaneously in the proposed approach.

The second approach fixes task mapping in all modes disallowing task migration as the existing approaches usually assume [67][71]. This technique is denoted as *Fixed*. The *Fixed* technique is implemented in the same GA framework as the proposed framework with disallowing task migration. Comparison with *Fixed* approach will show how task migration helps reduction of the resource requirement.

The third approach assumes that mode transition and task migration is performed in a blocking fashion. This technique is denoted as *Blocked*. The *Blocked* technique is also implemented in the same GA framework as the proposed framework, but uses a different formulation for the start offset of the next mode. Instead of Definition 5.4, $Lat(prev_mode) + MigCost(prev_mode, next_mode)$ is used for χ . Comparison with the *Blocked* approach will show how much benefit is expected by allowing mode-overlapped schedules for the resource requirement.

For all configurations in Table 4-4, we compared four techniques: *Base*, *Fixed*, *Blocked*, and *Proposed*. We assume that the minimum repetition count (*MRC*) for all modes in each example is set to the given value in Table 4-4 except the H.264 decoder application, since the mode transition pattern of the H.264 decoder is known and fixed (eg. *I-P-P-P-P-I-P-P-P-...*). Throughput constraints are set arbitrarily with considering the WCET of tasks. In the synthetic examples, vocoder, MP3 decoder, and printer receiver applications, the task migration cost is fixed to $MC(t)$ for all tasks. In H.264 decoder and lane detection applications, however, $MC(t)$ is scaled based on the actual task code size for all $t \in T$: the task migration cost of a task is computed as the product of $MC(t)$ values in Table 4-4.

Figure 4-14 shows the experimental results for all applications. The y-axis indicates the number of required processors. The results show that the *Proposed* approach requires no more processors than the other approaches in all applications.

Table 4-4. Configurations for experiments

	$MRC(m)$	$MC(t)$	$ThrConst$
<i>Example 1</i>	$MRC(m)=10$	$MC(t)=10$	1/130 iteration/time-unit
<i>Example 2</i>	$MRC(m)=10$	$MC(t)=10$	1/170 iteration/time-unit
<i>Example 3</i>	$MRC(m)=5$	$MC(t)=10$	1/300 iteration/time-unit
<i>Example 4</i>	$MRC(m)=5$	$MC(t)=50$	1/80 iteration/time-unit
<i>Example 5</i>	$MRC(m)=3$	$MC(t)=50$	1/2000 iteration/time-unit
<i>H.264 decoder</i>	$MRC(I)=1$ $MRC(P)=4$	$MC(t)=10$	1/5600 iteration/us
<i>Lane detection</i>	$MRC(m)=5$	$MC(t)=10$	1/700000 iteration/us
<i>Vocoder</i>	$MRC(m)=5$	$MC(t)=10000$	1/480000 iteration/cycle
<i>MP3 decoder</i>	$MRC(m)=3$	$MC(t)=10000$	1/5000000 iteration/time-unit
<i>Printer pipeline</i>	$MRC(m)=3$	$MC(t)=50$	1/1600 iteration/time-unit

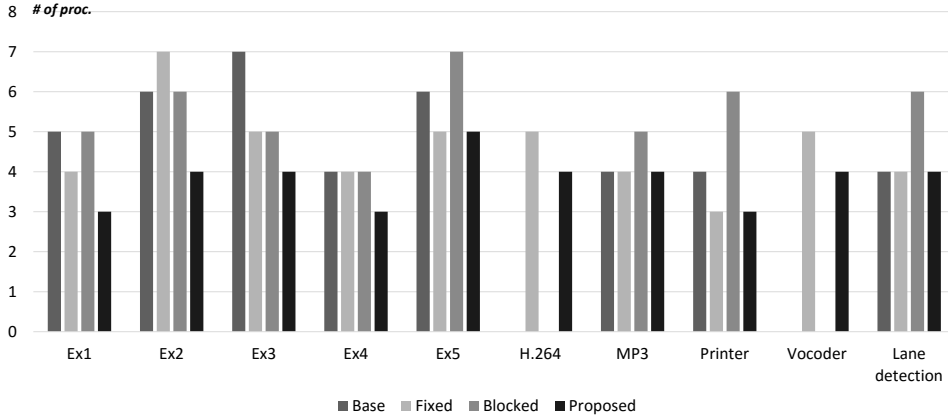


Figure 4-14. Comparison results in terms of the number of processors: *Base*, *Fixed*, *Blocked*, and *Proposed*

It is observed that the *Blocked* approach requires more processors than the other approaches that allow overlapped schedules during mode transition. In cases of H.264 decoder and vocoder applications, it could not find feasible solutions. It means that the blocking scheme degrades the overall throughput performance of an MMDF graph due to high mode transition delay.

Similarly to the *Blocked* approach, the *Base* approach requires more processors than the *Proposed* approaches. Because it constructs a static schedule of each mode without considering other modes, there is less chance of overlapping between the schedules during the mode transition. Therefore the mode transition delay is likely to be higher than the case of *Proposed* approach. It also could not find feasible solutions in cases of H.264 decoder and vocoder applications. Even though *Fixed* approach allows mode-overlapped schedule and constructs schedules of all modes simultaneously, it requires more processors than the *Proposed* approach in many cases. It is because it does not allow task migration among modes.

Table 4-5. Experimental results of *Proposed* approach

	$\max_{m \in Mode} \text{MaxTransition Delay}(m)$	$\min_{m \in Mode} \text{Thr Require}(m)$	<i>Output buffer size</i>	$MigCost_{total}$
<i>Example 1</i>	0 time-unit	1/130	2	20 time-unit
<i>Example 2</i>	20 time-unit	1/168	2	110 time-unit
<i>Example 3</i>	80 time-unit	1.284	2	40 time-unit
<i>Example 4</i>	10 time-unit	1/78	2	50 time-unit
<i>Example 5</i>	2622 time-unit	1/1126	2	1200 time-unit
<i>H.264 decoder</i>	660 us	1/5435	2	280 us
<i>Lane detection</i>	183000 us	1/663400	1	4500 us
<i>Vocoder</i>	88175 cycles	1/462364.8	2	340000 cycles
<i>MP3 decoder</i>	1412947 time-unit	1/4529017.6	2	340000 time-unit
<i>Printer pipeline</i>	804 time-unit	1/1332	2	1000 time-unit

In the lane detection application, there exists a dominant mode in which all tasks in an MMDF graph are executed. Since the dominant mode creates the critical path in all modes, if the mapping and scheduling result satisfies the throughput constraint in the dominant mode then results in the other modes automatically satisfy the throughput constraint. Hence, *Fixed*, *Base*, and *Proposed* approaches produce the same results for the application.

Table 4-5 presents the detailed experimental results from the *Proposed* approach in Figure 4-14. The table shows that the throughput which an application should satisfy becomes tighter than the given throughput constraint in Table 4-4 due to the mode transition delay. The table also presents the total task migration cost and the required output buffer sizes for benchmark applications.

4.6.2 Scalability of the Proposed Framework

Because the proposed framework is based on the genetic algorithm, its convergence speed depends on the size of solution space. As shown in Figure 9, the size of the solution space depends on the number of nodes and modes. So, we perform experiments for different configurations of these factors. Figure 13 shows the experimental results on the scalability of the proposed framework for synthetic examples. The results show that the number of nodes more contributes to the convergence speed than the number of modes.

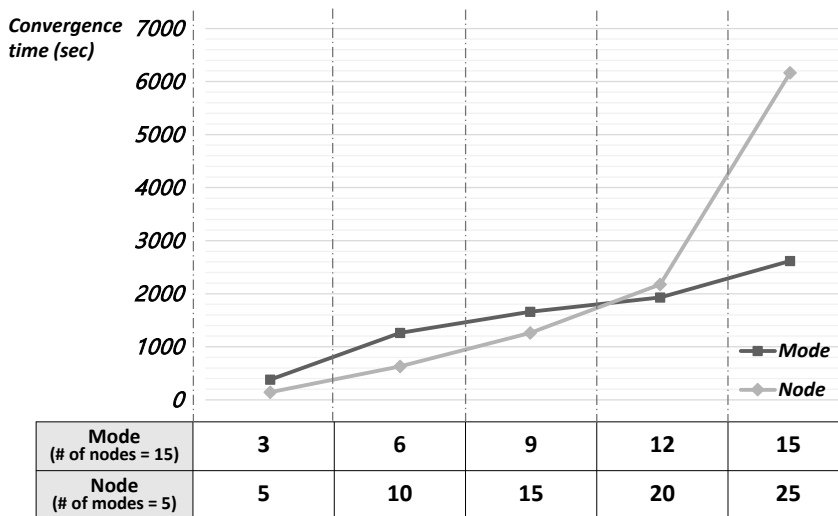


Figure 4-15. Experimental result for the scalability property

Chapter 5 Multiprocessor Code Generation for the Extended CIC Model

5.1 CIC translator

To execute a CIC task graph onto the target architecture, the CIC translator has been developed which synthesizes target executable code from the specified CIC task graph. The CIC translator synthesizes additional code which needs to be implemented with target dependent code such as task scheduling code and communication code.

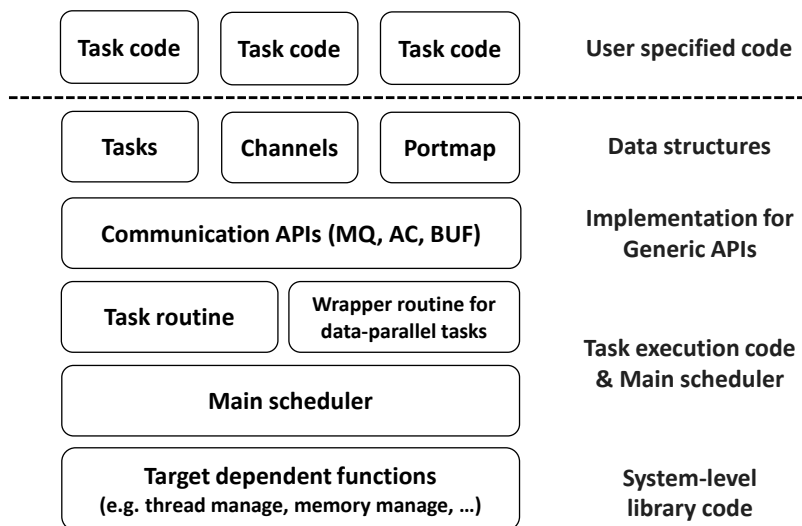


Figure 5-1. Code structure of the automatically generated code

In the previous CIC translator, it assumes that an application consists of a single task graph specified with KPN model. So, the CIC translator focuses on parallel code generation for the specified task graph on the heterogeneous system such as IBM CELL [106] and CPU+GPU [107] architectures. Therefore, it generates a scheduler code which invokes threads or processes for CIC computational tasks, and wrapper code for data parallel execution. Figure 5-1 shows a basic structure of the automatically generated code.

Because the previous CIC translator only considers parallel code generation for a single task graph, it needs to be extended to support extended features of both system-level and application-level dynamic behavior specification which are explained in Chapter 3. In this dissertation, how to support such extended features in the CIC translator will be introduced. Key contributions of the CIC translator extension are as follow:

- Application-level: multiprocessor code generation of an MTM-SDF graph considering the static scheduling results. Also it supports code generation for four different scheduling policies fully-static, self-timed, static-assignment, and fully-dynamic
- System-level: data structures for configurable task parameters, static scheduling result, and implementation of system request APIs

In the next chapter, how to implement each contribution will be explained in detail, respectively.

5.2 Code generation for application-level dynamism

In the extended CIC task model, a CIC computational task can be hierarchically composed, and a second-level task graph can be specified with an MTM-SDF model. To execute an MTM-SDF graph, a specified MTM information should be included to the generated code. Figure 5-2 shows automatically generated code for the specified MTM.

```
CIC_STATIC CIC_UT_MODE_MAP mode_map[] = {
    {0, "I-Frame"}, {1, "P-Frame"},
}
CIC_STATIC CIC_UT_INT_VAR int_var_map = {
    {0, "FrameVar", 0},
}
TRANSITION{
    CIC_T_INT FrameVar = GetVariableInt("FrameVar");
    CIC_MUTEX_LOCK(&mutex);
    switch(current_mode){
        case 0:
            if(FrameVar == 1){
                next_mode = 1;
                is_transition = CIC_V_TRUE;
            }
            ...
        case 1:
            ...
    }
    CIC_MUTEX_UNLOCK(&mutex);
    ...
}
```

Figure 5-2. Generated code for the specified MTM (*task_name.mtm*)

Because each mode of an MTM-SDF graph can be regarded as an SDF graph, task mapping and scheduling of each mode can be determined in compile-time. Therefore, the CIC translator should generate task execution code considering the static scheduling results.

Also, for the static scheduling results, four different scheduling policies [108] of the dataflow graph can be adopted as shown in Table 5-1.

Table 5-1. Scheduling policies for a dataflow graph

Strategy	Mapping	Scheduling	Timing
Fully-static	Compile-time	Compile-time	Compile-time
Self-timed	Compile-time	Compile-time	Run-time
Static-assignment	Compile-time	Run-time	Run-time
Fully-dynamic	Run-time	Run-time	Run-time

In fully-static policy, the compiler determines not only mapping and scheduling, but also exact firing time of each task. As shown in Figure 5-3 (a), its run-time execution will follow the static scheduling result, even though there exist execution time variation in run-time. Fully-static policy is very desirable for hard real-time systems because it guarantees that the execution of the task graph keeps the static scheduling result.

In self-timed policy, task mapping and scheduling are determined in compile-time, but its firing time is determined in run-time. At run-time, the processor waits for data to be available for the next actor in its scheduling list, and then fires that actor. As shown in Figure 5-3 (b), it shows higher utilization than fully-static policy, but it does not guarantee that its run-time execution always follows the static scheduling result.

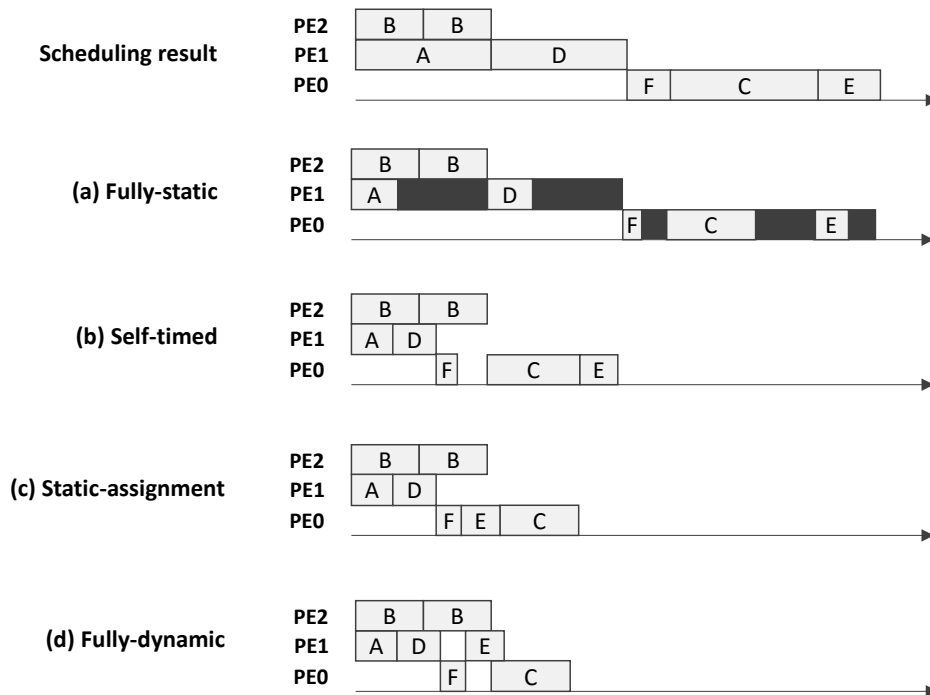


Figure 5-3. Run-time execution scenario for each scheduling policy

In static-assignment policy, a task is assigned to the processor at compile-time and a run-time scheduler invokes tasks assigned to the processor based on data availability. As shown in Figure 5-3 (c), it shows higher utilization than self-timed policy, but its run-time overhead may be larger than the self-timed policy, because it should determine task mapping as well as scheduling at run-time.

In fully-dynamic policy, tasks are mapped and scheduled at run-time only. When all input data for a given task are available, the task is assigned to an idle processor and fired. In general, it shows higher utilization than other policies, but it is not proper to real-time systems, because it cannot guarantee that the static scheduling result will be kept. So, the fully-dynamic policy is proper to implement casual systems in a best effort way.

Also, each scheduling policy can be implemented in different way depending on characteristics of the target system. In the dissertation, two different implementation methods are considered: thread version and function call version. In current implementation, fully-static and self-timed policies are implemented with function call version, and static-assignment and fully-dynamic policies are implemented with thread version. Depending on the implementation style, the CIC translator generates target executable code differently. Details will be explained in the next chapter.

5.2.1 Function call-style code generation (fully-static, self-timed)

For fully-static and self-timed policies, the CIC translator generates task execution code which invokes threads for each processor, because task execution order is fixed in each processor. Therefore, the CIC translator generates virtual task routines which include a function call sequence given by the static scheduling results. Because the static scheduling results are given for each task graph, and each scheduling result requires one or more processors, a group of virtual task routines will be generated for each task graph by the CIC translator. Then, a run-time scheduler should invoke threads for generated virtual task routines, not for specified task code which consists of TASK_INIT/GO/WRAPUP.

For this, the CIC translator generates code for data structures which include relation information between a task graph and generated virtual task routines. Also, for each virtual task routine, its assigned processor information should be included in the data structure. Figure 5-4 shows automatically generated code of data structures and virtual task routine for MTM-SDF graph in function call style.

Virtual task routines are generated automatically by the CIC translator as shown in Figure 5-4. In each virtual task routine, a function call sequence given by the

static scheduling result is generated. Because an MTM-SDF has a set of modes, it checks a current mode before execute the function call sequence, and then it calls functions depending on the current mode.

```

CIC_UT_TASK virtual_tasks[] = {
    ENTRY(11, "H264Dec_VIDEO_proc_0", ...);
    ENTRY(12, "H264Dec_VIDEO_proc_1", ...);
    ...
};

CIC_UT_VIRTUAL_TASK_TO_PROC_MAP virtual_task_to_core_map[] = {
    {11, 1}, {12, 2}, ...
};

// automatically generated task routine
CIC_T_VOID H264Dec_VIDEO_proc_0_Go(){
    ...
    mtms[mtm_index].UpdateCurrentMode("H264Dec_VIDEO_proc_0");
    mode = mtms[mtm_index].GetCurrentModeName
        ("H264Dec_VIDEO_proc_0");
    if(CIC_F_STRING_COMPARE(mode, "I_Frame") == 0){
        H264Dec_VIDEO_InterPredU_GO();
        H264Dec_VIDEO_InterPredV_GO();
        H264Dec_VIDEO_Deblock_GO();
    }
    else if(CIC_F_STRING_COMPARE(mode, "P_Frame") == 0){
        H264Dec_VIDEO_Deblock_Go();
        H264Dec_VIDEO_WriteFileH_Go();
    }
}

```

Figure 5-4. Data structure for MTM-SDF graphs and task execution routine considering the static scheduling result in the self-timed policy

```

// automatically generated task routine
CIC_T_VOID H264Dec_VIDEO_proc_0_Go(){
    mtms[mtm_index].UpdateCurrentMode("H264Dec_VIDEO_proc_4");
    if(CIC_F_STRING_COMPARE(mode, "I_Frame") == 0){
        clock_gettime(CLOCK_MONOTONIC, &start);
        H264Dec_VIDEO_InterPredU_GO();
        while(CIC_V_TRUE){
            clock_gettime(CLOCK_MONOTONIC, &end);
            diff = (end.tv_sec - start.tv_sec)*1000000
                  + ((end.tv_nsec - start.tv_nsec)/1000);
            if(300 <= diff)    break;
        }
        ...
    }
    else if(CIC_F_STRING_COMPARE(mode, "P_Frame") == 0){
        ...
    }
}

```

Figure 5-5. Task execution routine considering the static scheduling result in the fully-static policy

Especially, for the fully-static policy, a routine for firing time calculation is inserted between each function call as shown in Figure 5-5.

For each task graph, a group of threads are generated on each processor, not for each task. So, the main scheduler should invoke threads for virtual task routines when it needs to be executed. Figure 5-6 shows partial code of the main scheduler generated by the CIC translator. When a CIC computational task should be executed and it is hierarchically composed, the main scheduler invokes threads for VirtualTaskRoutine(). In VirtualTaskRoutine(), the thread is assigned to its mapped processor, and calls a virtual task routine in Figure 5-4.

```

CIC_UT_THREAD_FUNC_RET_TYPE VirtualTaskRoutine(...) {
    int processor_id = GetProcessorIdFromVirtualTaskId(task_id);
    CPU_SET(processor_id, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t)
                           , &cpuset);

    (*virtual_tasks[task_index].Init)(task_index);
    while(CIC_V_TRUE){
        (*virtual_tasks[task_index].Go)();
        ...
    }
    (*virtual_tasks[task_index].Wrapup)();
    ...
}
...
CIC_STATIC CIC_T_VOID ExecuteTasks(CIC_T_VOID){
    ...
    for(i=0; i<CIC_UV_NUM_VIRTUAL_TASKS; i++){
        CIC_T_INT parent_task_index;
        parent_task_index = GetTaskIndexFromTaskId
                           (virtual_tasks[i].parent_task_id);
        if(tasks[parent_task_index].state == STATE_RUN){
            CIC_F_THREAD_CREATE(&(virtual_tasks[i].thread)
                              , VirtualTaskRoutine, ...);
        }
    }
    ...
}

```

Figure 5-6. Task execution routine in the function call style

5.2.2 Thread-style code generation (static-assignment, fully-dynamic)

For static-assignment and fully-dynamic policies, the CIC translator generates task execution code which invokes threads for each task. Because, in the static-assignment policy, only the task mapping is fixed in compile-time, so it is a natural implementation which invokes a thread for each task. Then, a run-time scheduler will invoke a thread for each task, and assigns a processor depending on the static scheduling results. In the fully-dynamic policy, it just assigns a priority to each thread considering the static scheduling result.

```
CIC_UT_TASK tasks[] = {
    ENTRY(1, "H264Dec_VIDEO_ReadFileH", ...);
    ENTRY(2, "H264Dec_VIDEO_Decompile", ...);
    ...
};

CIC_TYPEDEF CIC_T_STRUCT{
    CIC_T_INT task_index;
    CIC_T_INT schedule_list[MAX_SCHED_NUM];
    CIC_T_CHAR* mode_list[MAX_MODE_NUM];
    CIC_T_INT core_map[MAX_SCHED_NUM][MAX_MODE_NUM];
} CIC_UT_TASK_TO_CORE_MAP;

CIC_UT_TASK_TO_PROC_MAP task_to_core_map[] = {
    {1, {4, }, {"I-Frame", "P-Frame", }, {{5, }, {6, }, }, },
    {2, {4, }, {"I-Frame", "P-Frame", }, {{4, }, {4, }, }, },
    ...
};
```

Figure 5-7. Data structure for MTM-SDF graphs in the thread style

Figure 5-7 shows data structure code for MTM-SDF graphs in static-assignment and fully-dynamic policies. Because one or more real-time constraints can be specified in a control task, and static schedule for each constraint will be different. Also, in an MTM-SDF graph, a task can be migrated between modes. Therefore, as shown in Figure 5-7 task-to-processor mapping information is given by a combination of schedule list and mode list for each task.

Figure 5-8 shows partial code of the main scheduler in the case of thread style. The main scheduler generates a thread for each task in the task graph which needs to be executed. In TaskRoutine() function, it checks its current schedule first, because its mapping and priority depend on the current schedule. The current schedule of the task graph will be set when SYS_REQ(SET_THROUGHPUT/LATENCY, ...) API is called. Because, an MTM-SDF graph consists of a set of modes and task mapping can be different in each mode, a mapped processor and priority of each thread will be determined at the start of every iterations. For the fully dynamic policy, task mapping will not be determined by the static scheduling result, but only the priority is assigned to each thread. Task mapping will be determined by the run-time scheduler of OS in the target platform at run-time.

```

CIC_UT_THREAD_FUNC_RET_TYPE TaskRoutine(...) {
    CIC_T_INT sched_id = GetCurrentScheduleId(task_index);
    ...
    TASK_INIT;
    while(CIC_V_TRUE){
        mtms[mtm_index].UpdateCurrentMode(task_name);
        mode_name = mtms[mtm_index].GetCurrentModeName(task_name);
        iter_count = mtms[mtm_index].GetTaskIterCount(task_name);

        // will not be generated in fully-dynamic policy
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(proc_id, &cpuset);
        pthread_setaffinity_np(pthread_self()
                                , sizeof(cpu_set_t), &cpuset);

        CIC_T_INT task_priority = GetTaskPriority(...);
        pthread_setschedprio(pthread_self(), task_priority);
        ...
        for(i=0; iter_count; i++)    TASK_GO;
    }
    TASK_WRAPUP;
}

CIC_STATIC CIC_T_VOID ExecuteTasks(CIC_T_VOID){
    for(i=0; i<CIC_UV_NUM_TASKS; i++){
        if(create_thread == CIC_V_TRUE && tasks[i].state == STATE_RUN)
            CIC_F_THREAD_CREATE(&(tasks[i].thread), TaskRoutine, ...);
    }
}

```

Figure 5-8. Task execution routine of an MTM-SDF graph in thread style

5.3 Code generation for system-level dynamism

For system-level dynamic behavior specification, the CIC control task is proposed which controls other CIC computational tasks using system request APIs. Not only the task execution, but also real-time constraints such as throughput and latency can be controlled by the CIC control task.

For task execution control, four system request APIs are supported: RUN_TASK/ STOP_TASK/SUSPEND_TASK/RESUME_TASK. Implementation of the system request APIs for task execution control depends on not only the target platform, but also the code generation style. In the function call style, threads for virtual task routines should be controlled when a system request API is called. On the other hand, in the thread style, threads for CIC computational tasks in the task graph should be controlled. For all code generation styles, basic implementation of system request APIs for task execution control is almost same except which threads should be controlled.

Figure 5-9 shows partial code for task execution control APIs. When a system request API for RUN_TASK is called, threads for the task graph will be created and executed, and when a system request API for STOP_TASK is called, threads for the task graph will be destroyed. Similarly, when a system request API for SUSPEND_TASK is called, threads of the task graph will be slept and wait on the conditional variable, and when a system request API for RESUME_TASK is called, the main scheduler will awake threads waiting on the conditional variable.

```

CIC_T_VOID RunCICTask(CIC_T_CHAR* task_name){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    for(i=0; i<CIC_UV_NUM_TASKS; i++){
        if(target_task_id == tasks[i].task_id){
            tasks[i].state = STATE_RUN;
            CIC_F_THREAD_CREATE(...);

            ...
        }
    }

CIC_T_VOID StopCICTask(CIC_T_CHAR* task_name){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    for(i=0; i<CIC_UV_NUM_TASKS; i++){
        if(target_task_id == tasks[i].task_id){
            tasks[i].state = STATE_STOP;
            CIC_F_THREAD_CANCEL(...);

            ...
        }
    }

CIC_T_VOID SuspendCICTask(CIC_T_CHAR* task_name){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    for(i=0; i<CIC_UV_NUM_TASKS; i++){
        if(target_task_id == tasks[i].task_id){
            tasks[i].state = STATE_WAIT;

            ...
        }
    }

CIC_T_VOID ResumeCICTask(CIC_T_CHAR* task_name){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    for(i=0; i<CIC_UV_NUM_TASKS; i++){
        if(target_task_id == tasks[i].task_id){
            tasks[i].state = STATE_RUN;
            CIC_F_COND_BROADCAST(...);

            ...
        }
    }
}

```

Figure 5-9. Implementation of system request APIs for task execution control

Also, a CIC control task can controls real-time constraint of the CIC computational task. In the HOPES framework, it parses task code of the CIC control task to extract specified real-time constraints. Then, the parsed constraint information is used to perform the multiprocessor scheduling explained in Chapter 4. Then, the scheduling results will be stored, and the CIC translator uses it to generate the static schedule code. Because a task graph can have multiple constraints depending on the state of the CIC control task, the CIC translator should generate a constraint-to-schedule table, and needs to support functions to change the current constraint of the task graph. Figure 5-10 shows a partial code for task constraint control.

```
CIC_T_VOID SetThroughputConstraint(CIC_T_CHAR* t_name,
CIC_T_CHAR* c_value, CIC_T_CHAR* c_unit){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    CIC_T_DOUBLE const = GetThroughputConst(c_value, c_unit);

    CIC_T_INT sched_id = GetSchedIdFromThroughputValue(const);
    SetCurrentSchedId(target_task_id, sched_id);
}

CIC_T_VOID SetLatencyConstraint(CIC_T_CHAR* t_name, CIC_T_CHAR*
c_value, CIC_T_CHAR* c_unit){
    CIC_T_INT target_task_id = GetTaskIdFromTaskName(task_name);
    CIC_T_DOUBLE const = GetLatencyConst(c_value, c_unit);

    CIC_T_INT sched_id = GetSchedIdFromLatencyValue(const);
    SetCurrentSchedId(target_task_id, sched_id);
}
```

Figure 5-10. Implementation of system request APIs for task constraint control

In summary, the CIC translator is extended to support both application-level and system-level dynamic behavior specification. In application-level, task execution code for an MTM-SDF graph is generated considering multiprocessor scheduling results. Also, it supports four different scheduling policies: fully-static, self-timed, static-assignment and fully-dynamic. Each policy is implemented with different code generation styles: function call and thread style. In system-level, implementation code for system request APIs is generated. Its implementation depends on not only the target platform, but also the code generation style.

Figure 5-11 shows a generated code structure for the extended CIC task model. Compare with Figure 5-1, code block highlighted with the gray color is additional generated by the CIC translator to support the extended features. To increase retargetability of the translated code, target dependent code part and target independent code part are separated. Target dependent functions are differently generated depending on the target platform, while the target independent code part can be used for various target platforms.

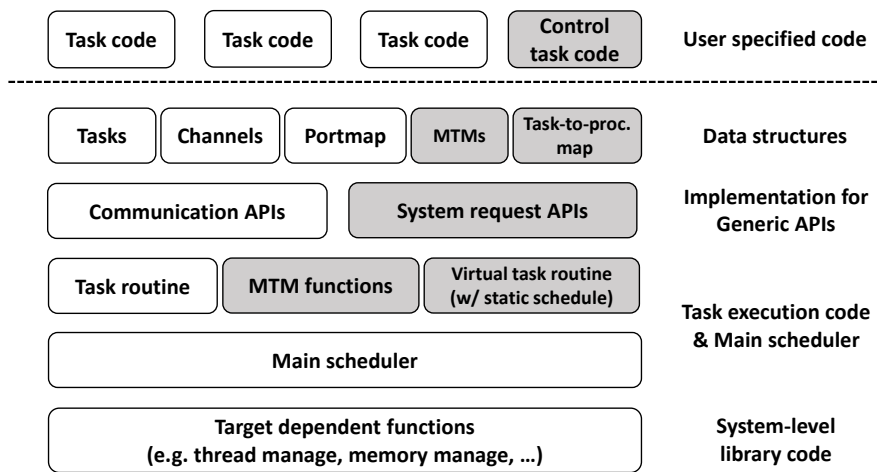


Figure 5-11. Code structure of the automatically generated code

5.4 Experimental results

To prove the viability of the generated code by the extended CIC translator, we compare the performance of four scheduling policies shown in Table 5-1 with the static scheduling result. Experiments are performed with three applications which are included in a multi-mode multimedia terminal example as shown in Figure 3-3: H.264 decoder, x264 encoder, and MP3 decoder. Especially, for H.264 decoder application, we perform experiments with two different configurations, *Video mode* and *Phone mode*, which have different throughput constraints. For x264 encoder example, 300 frame QCIF (176×144) video clip is used with 1/50000 iteration/msec throughput constraint. For H.264 decoder example, 2000 frame QCIF video clip is used with 1/5000 iteration/msec throughput constraint for *Video mode*, and 1/7000 iteration/msec throughput constraint for *Phone mode*, respectively. For

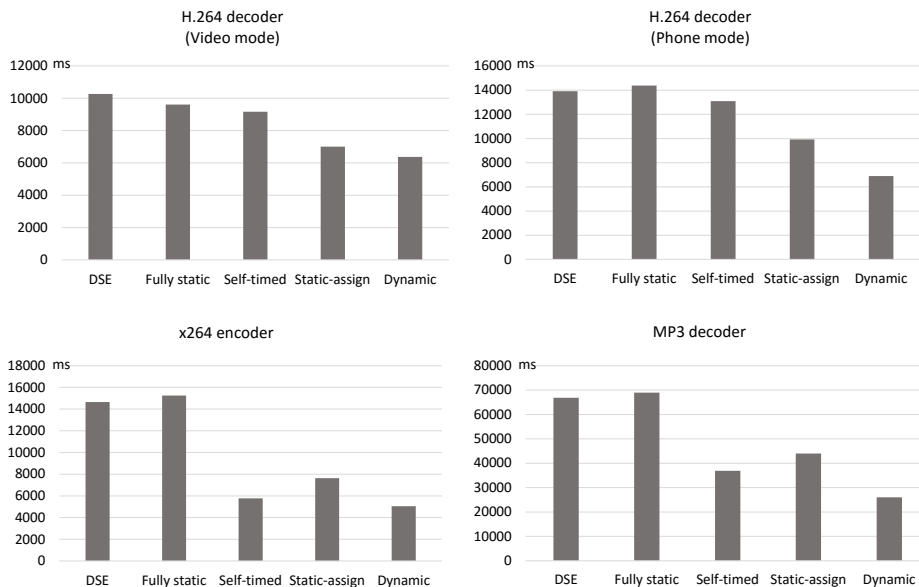


Figure 5-12. Comparison results between DSE results and four code generation policies for a multi-mode multimedia terminal example

MP3 decoder example, 33080 frame MP3 audio clip is used with 1/2500 iteration/msec throughput constraint.

Figure 5-12 shows the experimental results for each application. The x-axis denotes the execution time, and y-axis denotes scheduling policies. Basically the performance of each scheduling policy depends on execution time variation and run-time scheduling overhead, so which policy is better depends on the characteristics of the application and architecture. In fully-static policy, it shows almost same performance with the static scheduling result, because its run-time execution of mapping, scheduling, and firing time will follow the static scheduling result. So it is proper to implement predictable systems such as hard real-time systems. In self-timed and static-assignment policies, which policy shows better performance is varied depending on the characteristics of the application. Commonly, self-timed policy shows lower run-time overhead, but lower utilization, whereas static-assignment policy shows higher utilization, but higher run-time overhead. In fully-dynamic policy, it shows the best performance for all applications and configurations in Figure 5-12. However it cannot guarantee the performance of the static scheduling result, because it does not follow mapping, scheduling, and firing time of the static scheduling results. Therefore, it is proper to implement casual systems in a best-effort way such as entertainment system.

Chapter 6 Conclusion and Future Work

The dissertation covers how to design and implement modern embedded software which contain complex dynamic behavior as well as parallel processing. The proposed approach is based on model-based design framework and proposes the overall design flow from specification to automatic code generation.

First, it addresses the problem of how to specify the real-time embedded applications that have dynamic behavior on a multiprocessor platform. At first, it distinguishes two types of dynamism in the proposed model-based specification. To express multiple use cases of a system, it specifies each application as a dataflow task and the dynamic behavior as a control task that supervises the execution status of application tasks. To express multiple modes of operation for an application, on the other hand, MTM-SDF model is proposed. An MTM-SDF graph has a finite set of modes and each mode is specified by an SDF graph. Because the proposed specification is based on formal models such as dataflow model and finite state machine, dataflow model-based analysis can be performed in compile-time.

Next, it addresses the multiprocessor scheduling problem of an MTM-SDF graph allowing task migration with non-negligible mode transition delay. It observes that the mode transition delay should be considered in many streaming applications in which the mode transition occurs frequently, in order to satisfy the throughput constraint. Thus, the dissertation proposes a mapping/scheduling framework based on a genetic algorithm which schedules all SDF graphs simultaneously to minimize the number of processors while keeping the throughput constraint. Also, it proposes

the formulations to compute the required buffer size and the required throughput performance of the MMDF graph to satisfy the given throughput constraint of the system, by estimating the mode transition delay conservatively. To minimize the resource requirement, the proposed framework finds the maximally overlapped schedule which minimizes the mode transition delay. Experimental results confirm the superiority of the proposed technique over the other approaches.

Lastly, multiprocessor code generation technique is proposed for the extended task model. In application-level, it supports multiprocessor code generation for an MTM-SDF graph considering compile-time analysis results. Especially, it supports multiprocessor code generation for four different scheduling policies: fully-static, self-timed, static-assignment, and dynamic. In system-level, it generates implementation code for supported system request APIs which include thread management and data structures for static scheduling results and configurable properties.

The overall design flow is implemented and integrated to HOPES framework. As future work, the overall design flow should be verified for more real applications. Also, it needs to support various target platforms to show the viability of the proposed approach.

Bibliography

- [1] Wolf, W., A. Jerraya, A., and Martin, M., "Multiprocessor system-on-chip (MPSoC) technology," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, pp. 1701-1713, Oct. 2008.
- [2] J. E. D. E 2011, Failure mechanisms and models for semiconductor devices. <http://www.jedec.org/-standards-documents/docs/jep-122e>.
- [3] Nicolescu, G., and Mosterman, P. J., "Model-based design for embedded systems," CRC Press, 2009.
- [4] Lee, E. A. and Messerschmitt, D. G., "Synchronous data flow," Proceedings of the IEEE , vol.75, no.9, pp.1235,1245, Sept. 1987.
- [5] Kang, S., Ha, S., "A Multi-Objective Mapping-Scheduling Technique of Data Flow Application Considering Internal/External Data Parallelism," Journal of KIISE: Computing Practices and Letters, vol. 19, issue. 5, pp. 258-262. 2013.
- [6] Yang, H., Ha, S., "Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC," Design Automation and Test in Europe, pp. 69-74, Apr, 2009.

- [7] Stuijk, S., Geilen, M. C. W., Theelen, B. D., and Basten, T., "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," *Embedded Computer Systems (SAMOS)*, 2011 International Conference on, vol., no., pp.404,411, 18-21 July 2011.

- [8] Bhattacharya, B., and Bhattacharyya, S.S., "Parameterized dataflow modeling for DSP systems," *Signal Processing, IEEE Transactions on*, vol.49, no.10, pp.2408-2421, Oct 2001.

- [9] Bhattacharya, B., and Bhattacharyya, S. S., "Parameterized modeling and scheduling for dataflow graphs," MS thesis. research directed by Dept. of Electrical and Computer Engineering. University of Maryland, College Park, 1999.

- [10] Stuijk, S., Ghamarian, A. H., Theelen, B. D., Geilen, M. C. W., and Basten, T., "FSM-based SADF," MNEMEE internal report, TU Eindhoven, 2008.

- [11] Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., and Joo, Y., "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 12, Article No. 24., 2007.

- [12] Ha, S., Lee, C., Yi, Y., Kwon, S., and Joo, Y., "Hardware-software Codesign of Multimedia Embedded Systems: the PeaCE Approach," *The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, vol. 1, pp. 207-214, Aug, 2006.

- [13] Eidson, J., Lee, E. A., Matic, S., Seshia A. S., and Zou, J. "Distributed real-time software for cyber-physical systems," In Proceedings of the IEEE, vol.100, no.1, pp.45-59., 2012.
- [14] Zou, J., Matic, S., Lee, E. A., Feng T. H., and Derler, P. "Execution strategies for PTIDES, a programming model for distributed embedded systems," In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp.77-86., 2009.
- [15] Kwon, S., Kim, Y., Jeun, Y., Ha, S., and Paek, Y., "A Retargetable Parallel-Programming Framework for MPSoC," ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 13, pp. 1-18, Jul, 2008.
- [16] Park, H., Oh, H., and Ha, S., "Multiprocessor SoC design methods and tools," in IEEE Signal Processing Magazine, vol. 26, no. 6, pp. 72-79, November 2009.
- [17] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An integrated framework for MPSoC application parallelization," in Proc. 45th Design Automation Conf. (DAC), pp. 754–759, June, 2008.
- [18] Castrillon, J., Leupers, R., and Ascheid, G., "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs," in IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pp. 527-545, Feb. 2013.
- [19] Intel compiler [Online]. Available:
<https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers>

- [20] CriticalBlue multicore Cascade [Online]. Available:
http://www.criticalblue.com/criticalblue_products/multicore.shtml
- [21] OpenMP: API specification for parallel programming [Online].
Available: <http://openmp.org>
- [22] Dagum, L., and Menon, R., "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering vol. 5, no.1, pp. 46-55, 1998.
- [23] Lee, S., Min, S. J., and Eigenmann, R. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," ACM Sigplan Notices, vol. 44, no. 4, pp. 101-110, 2009.
- [24] IBM CELL Broadband Engine,
<http://www.ibm.com/developerworks/power/cell>
- [25] O'Brien, K., O'Brien, K., Sura, Z., Chen, T., and Zhang, T.,
"Supporting OpenMP on cell," Int. J. Parallel Programming, vol. 36,
no. 3, pp. 289–311, June 2008.
- [26] Labarta, Jesus., "StarSS: A programming model for the multicore era."
PRACE Workshop'New Languages & Future Technology Prototypes'
at the Leibniz Supercomputing Centre in Garching (Germany). 2010.
- [27] Intel® Cilk Plus [Online]. Available:
<https://software.intel.com/en-us/intel-cilk-plus>
- [28] Message passing interface (MPI) [Online]. Available:
<http://www.mpi-forum.org>

- [29] Gropp, W., Lusk, E., and Skjellum, A., "Using MPI: portable parallel programming with the message-passing interface," vol. 1. MIT press, 1999.
- [30] Kepner, J., "MatlabMPI," J. Parallel Distrib. Comput., vol. 64, no. 8, pp. 997- 1005, Aug. 2004.
- [31] Nvidia CUDA: General-purpose parallel computing architecture [Online].Available: <http://www.nvidia.com/cuda>
- [32] Sanders, J., and Kandrot, E., "CUDA by example: an introduction to general-purpose GPU programming," Addison-Wesley Professional, 2010.
- [33] Khronos OpenCL: The open standard for parallel programming of heterogeneous systems [Online]. Available: <http://www.khronos.org/opencvl>
- [34] Stone, J. E., Gohara, D., and Shi, G. "OpenCL: A parallel programming standard for heterogeneous computing systems." Computing in science & engineering, vol. 12, issue. 1-3, pp. 66-73, 2010.
- [35] Lee, E. A., Neuendorffer, S., and Zhou, G., "System Design, Modeling, and Simulation using Ptolemy II," Ptolemy.org, 2014.
- [36] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming, vol. 8, pp. 231-274, 1987.
- [37] Kahn, G., "The semantics of a simple language for parallel programming," Proceedings of IFIP Congress 74, pp. 471,475, 1974.

- [38] S. Bhattacharyya, S. et al. "Dynamic dataflow graphs," In Handbook of Signal Processing Systems. 2nd edition, 2013
- [39] Buck, J. T., "Scheduling dynamic dataflow graphs with bounded memory using the token flow Model," Ph.D. Thesis, U.C.Berkeley, CA., 1993.
- [40] Kock, E. A., Smits, W. J. M., Wolf, P., Brunel, J. Y., Kruijtzter, W. M., Lieverse, P., Vissers, K. A., and Essink, G., "YAPI: Application modeling for signal processing systems," In Proceedings of Design Automation Conference, pp.402-405., 2000.
- [41] Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., and Deprettere. E., "Daedalus: toward composable multimedia mp-soc design," In DAC '08: Proceedings of the 45th annual Design Automation Conference, pages 574-579, ACM, New York, NY, USA, 2008.
- [42] Thompson, M., Nikolov, H., Stefanov, T., D. Pimentel, A., Erbas, C., Polstra, S., and F. Deprettere, E., "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," In Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 9-14, 2007.
- [43] Verdoolaege, S., Nikolov, H., and Stefanov, T., "PN: a tool for improved derivation of process networks," EURASIP Journal on Embedded Systems, vol. 2007, Article ID 75947, 2007.
- [44] Stuijk, S., Basten, T., Geilen, M. C. W., and Corporaal, H., "Multiprocessor resource allocation for throughput-constrained

synchronous dataflow graphs." Proceedings of the 44th annual Design Automation Conference. ACM, 2007.

- [45] Bonfietti, A., Benini, L., Lombardi, M., and Milano, M., "An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms." Proceedings of the Conference on Design, Automation and Test in Europe. European Design and Automation Association, 2010.
- [46] Thies, W., Karczmarek, M., and P. Amarasinghe, S., "StreamIt: A Language for Streaming Applications," In Proceedings of the 11th International Conference on Compiler Construction (CC '02), R. Nigel Horspool (Ed.). Springer-Verlag, London, UK, UK, 179-196, 2002.
- [47] A. Bamakhrama, M., Teddy Zhai, J., Nikolov, H., and Stefanov, T., "A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems," In DATE 2012: Proceedings of the 15th Design, Automation, and Test in Europe conference, March 12-16, 2012.
- [48] Gedae [Online]. Available: <http://www.gedae.com>
- [49] Pelcat, M., Piat, J., Wipliez, M., Aridhi, S., Nezan, J., "An Open Framework for Rapid Prototyping of Signal Processing Applications". EURASIP Journal on Embedded Systems, 2009.
- [50] Girault, A., Lee, B., and Lee, E. A., "Hierarchical finite state machines with multiple concurrency models," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 742–760., 1999.

- [51] Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., and Teich, J., "FunState-an internal design representation for Codesign," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 4, pp. 524 – 544., 2001.
- [52] Harel, D., and Naamad, A., "The STATEMATE semantics of statecharts," ACM Trans. Software Eng. Meth., Vol. 5, Article No. 4., 1996.
- [53] Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubuhr, T., Deyhle, A., Hadert, D., and Teich, J., "A SystemC-Based Design Methodology for Digital Signal Processing Systems," EURASIP Journal on Embedded Systems, Vol. 2007, pp. 22., 2007.
- [54] Stuijk, S., Ghamarian, A., Theelen, B., Geilen, M., and Basten, T., "FSM-based SADF", Technical report, Eindhoven University of Technology, Department of Electrical Engineering, 2008.
- [55] Theelen, B. D., Geilen, M., Basten, T., Voeten, J., Gheorghita, S. V., and Stuijk, S. "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," In Proceedings of International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp.185-194, 2006.
- [56] Stuijk, S., Geilen, M. C. W., Theelen, B. D., and Basten, T., "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," Embedded Computer Systems (SAMOS), 2011 International Conference on, vol., no., pp.404, 411, 18-21 July 2011.

- [57] Geilen, M., and Basten, T. "Reactive process networks," In Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT), pp.137-146., 2004.
- [58] Geilen, M., and Basten, T. "Kahn process networks and a reactive extension," Handbook of Signal Processing Systems. pp. 967-1006., 2010.
- [59] Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S., and Thiele, L., "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," In Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems (CASES '12), pp.71,80, 2012.
- [60] Savage, J. E. "Models of computation. Exploring the Power of Computing," 1998.
- [61] Thiele, L., Bacivarov, I., Haid, W., and Kai Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on, vol., no., pp.29,40, 10-13 July 2007.
- [62] Ecker, W., Müller, W., and Dömer, R., "Hardware-dependent Software." Hardware-dependent Software. Springer Netherlands, 2009. 1-13.
- [63] Thiele, L., Chakraborty, S., Gries, M., and Kunzli, S., "A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures," In Proc. 39th Design Automation Conference (DAC 2002), pp. 880–885, New Orleans, LA, USA, June 2002.

- [64] Kim, D., Kim, M., and Ha, S., "A Case Study of System Level Specification and Software Synthesis of Multi-mode Multimedia Terminal," Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2003.
- [65] Geilen, M., and Stuijk, S., "Worst-case Performance Analysis of Synchronous Dataflow Scenarios," In Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and System Synthesis (CODES/ISSS '10). ACM, New York, NY, USA, pp.125-134, 2010.
- [66] Damavandpeyma, M., Stuijk, S., Geilen, M., Basten, T., and Corporaal, H. "Parametric throughput analysis of scenario-aware dataflow graphs." Computer Design (ICCD), 2012 IEEE 30th International Conference on. IEEE, 2012.
- [67] Stuijk, S., Geilen, M., and Basten, T., "A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour," In Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on. pp.548–555, 2010.
- [68] Bastos, J., Stuijk, S., Voeten, J., Schiffelers, R., Jacobs, J., and Corporaal, H., "Modeling resource sharing using FSM-SADF." Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on. IEEE, 2015.
- [69] Damavandpeyma, M., Stuijk, S., Basten, T., Geilen, M., and Corporaal, H., "Throughput-constrained DVFS for scenario-aware dataflow graphs," In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th. pp.175–184, 2013.

- [70] Moreira, O., and Corporaal, H., "Mode-Controlled Data Flow." Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor. Springer International Publishing, pp. 117-152, 2014.
- [71] Moreira, O., "Temporal analysis and scheduling of hard real-time radios running on a multiprocessor," Ph.D. Dissertation. ser. PHD Thesis, Technische Universiteit Eindhoven, 2012.
- [72] Wiggers, M., Bekooij, M., and Smit, G., "Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication," In Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE. pp.183–194, 2008.
- [73] Wiggers, M., Bekooij, M., and Smit, G., "Buffer Capacity Computation for Throughput-constrained Modal Task Graphs," Article 17, Jan, 2011.
- [74] Zhai, J. "Adaptive streaming applications: analysis and implementation models," Ph.D. Dissertation. Leiden Embedded Research Center, Faculty of Science (LERC), Leiden Institute of Advanced Computer Science (LIACS), Leiden University. 2015.
- [75] Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete. J., "Cyclo-static dataflow," IEEE Trans. Signal Processing, vol. 44, no. 2, pp. 397–408, Feb. 1996.
- [76] Verdoolaege, Sven. "Polyhedral process networks." Handbook of Signal Processing Systems. Springer New York, pp. 1335-1375, 2013.

- [77] Bamakhrama, M., and Stefanov, T., "Hard-Real-Time Scheduling of Data-Dependent Tasks in Embedded Streaming Applications," In EMSOFT '11: Proceedings of the 9th ACM International Conference on Embedded Software, pp. 195-204, October 9-14, 2011.
- [78] Bebelis, V., Fradet, P., Girault, A., and Lavigueur, B., "BPDF: A Statically Analyzable DataFlow Model with Integer and Boolean Parameters," In Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT '13). IEEE Press, Piscataway, NJ, USA, Article 3, 10 pages, 2013.
- [79] Bempelis, E. "Boolean Parametric Data Flow Modeling-Analyses -Implementation," Diss. Université Grenoble Alpes, 2015.
- [80] Deb, K., and Kalyanmoy, D., "Multi-Objective Optimization Using Evolutionary Algorithms," John Wiley & Sons, Inc., 2001.
- [81] Oh, H., and Ha, S., "Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints," Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on, vol., no., pp.133,138, 2002.
- [82] Shabbir, A., Stuijk, S., Kumar, A., Corporaal, H., and Mesman, B., "An MPSoC design approach for multiple use-cases of throughput constrained applications," In Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11), 2011.
- [83] Lee, C., Kim, S., Oh, H., and Ha, S., "Failure-Aware Task Scheduling of Synchronous Data Flow Graphs Under Real-Time Constraints," *Journal of Signal Processing Systems*, vol. 72, issue. 2, pp. 201,212, May 2013.

- [84] Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffmann, H., and Amarasinghe, S. "StreamIt: A Compiler for Streaming Applications," 2002.
- [85] Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., Meli, A. S., and Lamb, A. "A stream compiler for communication-exposed architectures," ACM SIGPLAN Notices. vol. 37. no. 10. ACM, 2002.
- [86] Gordon, M. I., Thies, W., and Amarasinghe, S., "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs." ACM SIGOPS Operating Systems Review, vol. 40, no.5, pp. 151-162, 2006.
- [87] Haid, S., Schor, L., Huang, K., Bacivarov, I., and Thiele, L., "Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs," 2009 IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia, Grenoble, pp. 35-44, 2009.
- [88] Xeon Phi Code Generator - Distributed Application Layer,
http://www.dal.ethz.ch/index.php?title=Xeon_Phi_Code_Generator
- [89] Intel SCC Processor Code Generator - Distributed Application Layer,
http://www.dal.ethz.ch/index.php?title=Intel_SCC_Processor_Code_Generator
- [90] Distributed Linux Code Generator - Distributed Application Layer,
http://www.dal.ethz.ch/index.php?title=Distributed_Linux_Code_Generator
- [91] Gedae Idea Compiler, http://www.gedae.com/idea_compiler.html

- [92] Desnos, K., Heulot, J., "PiSDF: Parameterized & Interfaced Synchronous Dataflow for MPSoCs Runtime Reconfiguration," 1st Workshop on Methods and Tools for Dataflow Programming (METODO), proceedings, 2014.
- [93] Pelcat, M., Nezan, J., Piat, J., Croizer, J., Aridhi, S., "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems". DASIP Sophia Antipolis, 2009.
- [94] Zhou, Z., Desnos, K., Pelcat, M., Nezan, J., Plishker, W., Bhattacharyya, S., "Scheduling of Parallelized Synchronous Dataflow Actors," SoC13, Tampere, Finland, 2013.
- [95] Piat, J., Bhattacharyya, S., Pelcat, M., Raulet, M., "Multi-Core Code Generation From Interface Based Hierarchy," DASIP Sophia Antipolis, 2009.
- [96] Richardson, I. E., "H. 264 and MPEG-4 video compression: video coding for next-generation multimedia," John Wiley & Sons, 2004.
- [97] Kim, D., and Ha, S., "Static analysis and automatic code synthesis of flexible FSM model," Proceedings of the 2005 Asia and South Pacific Design Automation Conference. ACM, 2005.
- [98] Man, K. F., Tang, K. S., & Kwong, S., "Genetic algorithms: concepts and applications," IEEE transactions on Industrial Electronics, vol. 43, issue. 5, pp. 519-534., 1996.
- [99] Choi, J., Kang D., and Ha, S., "Conservative Modeling of Shared Resource Contention for Dependent Tasks in Partitioned Multi-Core

Systems," Design Automation and Test in Europe, pp. 181-186, Mar, 2016.

- [100] Pellizzoni, R., Schranzhofer, A., Chen, J., Caccamo, M., and Thiele, L., "Worst Case Delay Analysis for Memory Interference in Multicore Systems," Proceedings of Design, Automation & Test in Europe Conference & Exhibition, pp. 741–746, Mar, 2010.
- [101] Zitzler, E., Laumanns, M., and Thiele, L., "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, May 2001.
- [102] Benini, L., and De Micheli, G., "Networks on chips: A new SoC paradigm. Computer," vol. 35, issue. 1, pp. 70–78, Jan. 2002.
- [103] Thiele, L., Chakraborty, S., and Naedele, M., "Real-time calculus for scheduling hard real-time systems," *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol.4, no., pp.101,104 vol.4, 2000.
- [104] Geilen, M., Stuijk, S., "Worst-case performance analysis of Synchronous Dataflow scenarios," Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on., pp. 125-134, 2010.
- [105] Yang, Y., "Exploring resource/performance trade-offs for streaming applications on embedded multiprocessors," Ph.D. Dissertation. Ph. D. thesis, Eindhoven University of Technology, Eindhoven. 2012.

- [106] Kim, K., Lee, J., Park, H., and Ha, S., "Automatic H.264 Encoder Synthesis for the Cell Processor from a Target Independent Specification," in Proc. 6th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), pp. 95-100, Oct, 2008.

- [107] Jung, H., Yi, Y., and Ha, S., "Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU architectures," Parallel Processing and Applied Mathematics 2011, Sep, 2011.

- [108] Lee, E. A., and Ha, S., "Scheduling strategies for multiprocessor real-time DSP," Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond' (GLOBECOM), 1989. IEEE, Dallas, TX, pp. 1279-1283 vol.2, 1989.

초록

하나의 칩에 집적되는 프로세서의 개수가 많아지고, 많은 기능들이 통합됨에 따라, 연산량의 변화, 서비스의 품질, 예상치 못한 시스템 요소의 고장 등과 같은 다양한 요소들에 의해 시스템의 상태가 동적으로 변화하게 된다. 반면에, 본 논문에서 주된 관심사를 가지는 스마트 폰 장치에서 주로 사용되는 비디오, 그래픽 응용들의 경우, 계산 복잡도가 지속적으로 증가하고 있다. 따라서, 이렇게 동적으로 변하는 행위를 가지면서도 병렬성을 내제한 계산 집약적인 연산을 포함하는 복잡한 시스템을 구현하기 위해서는 체계적인 설계 방법론이 고도로 요구된다.

모델 기반 방법론은 병렬 임베디드 소프트웨어 개발을 위한 대표적인 방법 중 하나이다. 특히, 시스템 명세, 정적 성능 분석, 설계 공간 탐색, 그리고 자동 코드 생성까지의 모든 설계 단계를 지원하는 병렬 임베디드 소프트웨어 설계 환경으로서, HOPES 프레임워크가 제시되었다. 다른 설계 환경들과는 다르게, 이기종 멀티프로세서 아키텍처에서의 일반적인 수행 모델로서, 공통 중간 코드 (CIC) 라고 부르는 “프로그래밍 플랫폼”이라는 새로운 개념을 소개하였다. CIC 태스크 모델은 프로세스 네트워크 모델에 기반하고 있지만, SDF 모델로 구체화될 수 있기 때문에, 병렬 처리뿐만 아니라 정적 분석이 용이하다는 장점을 가진다. 하지만, SDF 모델은 응용의 동적인 행위를 명세할 수 없다는 표현상의 제약을 가진다.

이러한 제약을 극복하고, 시스템의 동적 행위를 응용 외부와 내부로 구분하여 명세하기 위해, 본 논문에서는 데이터 플로우와 유한상태기 (FSM) 모델에 기반하여 확장된 CIC 태스크 모델을 제안한다. 상위 수준에서는, 각 응용은 데이터 플로우 태스크로 명세되며, 동적 행위는 응

용들의 수행을 감독하는 제어 태스크로 모델 된다. 데이터 플로우 태스크 내부에는, 유한상태기 기반의 SADF 모델과 유사한 형태로 동적 행위가 명세 된다; SDF 태스크는 복수개의 행위를 가질 수 있으며, 모드 전환기 (MTM)이라고 불리는 유한 상태기의 테이블 형태의 명세를 통해 SDF 그래프의 모드 전환 규칙을 명세 한다. 이를 MTM-SDF 그래프라고 부르며, 복수 모드 데이터 플로우 모델 중 하나로 구분된다. 응용은 유한한 행위 (또는 모드)를 가지며, 각 행위 (모드)는 SDF 그래프로 표현되는 것을 가정한다. 이를 통해 다양한 프로세서 개수에 대해 단위시간당 처리량을 최대화하는 컴파일-시간 스케줄링을 수행하고, 스케줄 결과를 저장할 수 있도록 한다.

또한, 복수 모드 데이터 플로우 그래프를 위한 멀티프로세서 스케줄링 기법을 제시한다. 복수 모드 데이터 플로우 그래프를 위한 몇몇 스케줄링 기법들이 존재하지만, 모드 사이에 태스크 이주를 허용한 기법들은 존재하지 않는다. 하지만 태스크 이주를 허용하게 되면 자원 요구량을 줄일 수 있다는 발견을 통해, 본 논문에서는 모드 사이의 태스크 이주를 허용하는 복수 모드 데이터 플로우 그래프를 위한 멀티프로세서 스케줄링 기법을 제안한다. 유전 알고리즘에 기반하여, 제안하는 기법은 자원 요구량을 최소화하기 위해 각 모드에 해당하는 모든 SDF 그래프를 동시에 스케줄 한다. 주어진 단위 시간당 처리량 제약을 만족시키기 위해, 제안하는 기법은 각 모드 별로 실제 처리량 요구량을 계산하며, 처리량의 불규칙성을 완화하기 위한 출력 버퍼의 크기를 계산한다.

명세된 태스크 그래프와 스케줄 결과로부터, HOPES 프레임워크는 대상 아키텍처를 위한 자동 코드 생성을 지원한다. 이를 위해 자동 코드 생성기는 CIC 태스크 모델의 확장된 특징들을 지원하도록 확장되었다. 응용 수준에서는 MTM-SDF 그래프를 주어진 정적 스케줄링 결과를 따르

는 멀티프로세서 코드를 생성하도록 확장되었다. 또한, 네 가지 서로 다른 스케줄링 정책 (fully-static, self-timed, static-assignment, fully-dynamic)에 대한 멀티프로세서 코드 생성을 지원한다. 시스템 수준에서는 지원하는 시스템 요청 API에 대한 실제 구현 코드를 생성하며, 정적 스케줄 결과와 태스크들의 제어 가능한 속성들에 대한 자료 구조 코드를 생성한다.

복수 모드 멀티미디어 터미널 예제를 통한 기초적인 실험들을 통해, 제안하는 방법론의 타당성을 보인다.

주요어 : 모델기반 설계 방법론, 동기화된 데이터 흐름 그래프, 복수 모드 데이터
흐름 그래프, 멀티프로세서 스케줄링, 모드 전환 지연, 자동 코드 합성, 병렬 코드
생성

학 번 : 2012-30230